MULTIPROCESSOR ANALYSIS OF POWER SYSTEM TRANSIENT STABILITY

A thesis

presented for the degree

of

Doctor of Philosophy in Electrical Engineering

in the

University of Canterbury,

Christchurch, New Zealand

by

M.I. PARR B.E.(Hons)

University of Canterbury

1983

CONTENTS

	Page
List of Principal Symbols	vii
Abstract	viii
Acknowledgements	ix
CHAPTER 1 INTRODUCTION	1
CHAPTER 2 EXECUTION OF A TRANSIENT STABILITY ANALYSIS PROGRAM	9
2.1 Introduction	9
2.2 Transient Stability Concepts	10
2.2.1 Steady State Stability	11
2.2.2 Transient Stability	12
2.3 Transient Stability Analysis	15
2.3.1 Approaches to Solution	16
2.4 Serial Execution Characteristics	18
2.4.1 Parallelism and Dependence	19
2.4.2 Computational Blocks	22
2.4.3 Categorisation of Blocks	22
2.4.4 Distribution of Processing Time	24
2.4.5 Identification of the Area Most Suited to Parallel Execution	26
2.5 Parallel Execution within the Central Loop	27
2.5.1 Generator models	27
2.5.2 Network models	28
2.5.3 A Bottleneck	29
2.6 Summary	29
CHAPTER 3 MULTIPROCESSOR TYPES AND THEIR SUITABILITY TO TRANSIENT STABILITY ANALYSIS	30
3.1 Introduction	30
3.2 Classification of Multiprocessors	31
3.2.1 SISD - Serial processors	31
3.2.2 SIMD Processors	33
3.2.3 MIMD Processors	34
3.2.4 Pipelined Processors	35
3.2.5 Data Flow Processors	37

3.3 Selection of Appropriate Form for Transient Stability Analysis 39
3.3.1 Identification of Requirements 39
3.3.2 Investigation of an SIMD Implementation 40
3.4 Conclusions 43

CHAPTER 4 PARALLEL SOLUTION OF LINEAR EQUATIONS 44
4.1 Introduction 44
4.2 The Problem Specified 44
4.3 Algorithmic Developments 48
4.3.1 The Sequential Method 48
4.3.2 Block Oriented Schemes 48
4.3.3 Elemental Approaches 50
4.3.3.1 EPRI Methods 52
4.4 The PBIF Algorithm 54
4.4.1 Task Identification 55
4.4.2 Scheduling 57
4.4.3 Process Definition 58
4.4.3.1 Use of Semaphores 61
4.4.3.2 Coding 61
4.4.3.3 Memory Requirements 63
4.5 Conclusion 65

CHAPTER 5 EXISTING MIMD MULTIPROCESSING SYSTEMS 66
5.1 Introduction 66
5.2 Bus Structures 67
5.2.1 Variety of Structures 68
5.2.2 Criteria for Comparison of Structures 69
5.2.3 Specialisation and Polymorphism 70
5.3 Structures Adopted in Some Existing and Proposed Systems 71
5.3.1 Structural Proposals by Fong 72
5.3.2 PAPROS 73
5.3.3 MOPPS 75
5.3.4 DEMOS 76
5.3.5 CM* 77
5.3.6 C.mmp 79
5.3.7 FMP 79
5.4 Malfunction Detection and Recovery 82

5.5 Salient Features of Existing Systems 83

CHAPTER 6 HARDWARE REQUIREMENTS AND IMPLEMENTATION OF THE UCMP SYSTEM 87

6.1 Introduction 87

6.2 Functional Requirements, Options, and Component Selection 89

6.2.1 Numerical Representation 90

6.2.1.1 Wordlength Required 91

6.2.2 Memory 95

6.2.2.1 Global 96

6.2.2.2 Local 97

6.2.3 Number of Processors and Bus Structure 97

6.2.4 Processor Capability 101

6.2.4.1 Addressing 101

6.2.4.2 Program Functions 102

6.2.5 Support 103

6.3 UCMP System Description 105

6.3.1 Bus Structure 105

6.3.1.1 Processing Elements and Local Memories 105

6.3.1.2 Priority Resolution 109

6.3.1.3 Malfunction Considerations 110

6.3.2 Address Structure 110

6.3.3 Control Structure 114

6.3.4 Basic Operation 115

6.3.5 Operation Support Options 116

6.3.5.1 MDS Based Operation 119

6.3.5.2 VAX Based Operation 120

6.4 Components 122

6.4.1 Processing Elements 122

6.4.1.1 iSBC 86/12 Single Board Computer 123

6.4.1.2 UC/86 Single Board Computer 126

6.4.2 Memory 128

6.4.3 Backplane 128

6.4.4 VAX Computer Interface 130

6.4.4.1 Input/Output 132

6.4.4.2 DMA 134

6.5 Summary 135

CHAPTER 7 UCMP SYSTEM SOFTWARE UTILITIES AND OPERATION 136

7.1 Introduction 136

7.2 Programming Environment 138

7.2.1 Address Structure 138

7.2.2 Hierachy 140

7.3 Code Preparation, Debugging, and Execution 141

7.3.1 Data Flow 141

7.3.2 Development Cycle 143

7.3.2.1 Code Preparation 144

7.3.2.2 Serial Debugging 145

7.3.2.3 Parallel Debugging 145

7.3.2.4 Execution and Performance Evaluation 146

7.3.2.5 Data Changes 147

7.4 Utility Software 147

7.4.1 File Formats 148

7.4.2 MDS Resident Utilities 149

7.4.2.1 Languages 149

7.4.2.2 Linkage and Location 150

7.4.2.3 RELOC - Load Address Relocater 150

7.4.2.4 OBJASC and ASCOBJ - File Format Transformers 152

7.4.3 VAX Resident Utilities 152

7.4.3.1 UCTS Program 152

7.4.3.2 CRBACK - Parallel Bifactorisation Vector Formation Utility 153

7.4.3.3 CRASC - ASC Format File Creation Utility 153

7.4.3.4 INTUCMP - Interactive Program 154

7.4.4 Stand Alone Utilities 155

7.4.4.1 8086 Monitor and Loader 155

7.4.4.2 8085 Monitor and Loader 156

7.4.4.3 System Oriented Service Programs 156

CHAPTER 8 SIMULATION OF THE PARALLEL EXECUTION OF THE SOLUTION OF LINEAR EQUATIONS 159

8.1 Introduction 159

8.2 Principle of Operation 160

8.3 Overheads 161

8.3.1 General Categories 161

8.3.2 PBIF Algorithm 164

8.3.3 Models Included 164

8.3.4 Categories for Recording 166

8.4 Benchmark Processor and Network Choice 166

8.5 Performance Measured 167

8.5.1 Overall System Performance 171

8.5.2 Distribution of Inefficiency Among Overheads 172

8.6 Conclusions 173

CHAPTER 9 HARDWARE PERFORMANCE - THE SOLUTION OF LINEAR EQUATIONS 174

9.1 Introduction 174

9.2 Program Implementation 175

9.3 Comparison of Performance of Hardware with that Predicted by Simulation 175

9.4 Various Practical Power Systems Modelled 177

9.5 Nodal Distribution Ill-Conditioning 178

9.6 Variations and Sensivity to Program Parameters 185

9.6.1 Search Length 187

9.6.2 Semaphore Examination 189

9.7 Bus Conflicts 190

9.7.1 Bandwidth and Conflict 190

9.7.2 Modelling Bus Conflict 192

9.7.3 Processor Modelling Techniques 192

9.7.4 Method Implemented to Induce Bus Conflict 193

9.7.5 Results 196

9.8 Experience with the UCMP Hardware 199

9.8.1 Non-Constant Execution Times 199

9.8.2 Priority Resolution Problems 200

9.9 Conclusions 201

CHAPTER 10 HARDWARE PERFORMANCE - INCLUSION OF GENERATOR MODELS 202

10.1 Introduction 202

10.2 Constrained Ideal Performance 203

10.3 Execution Sequencing 204

10.3.1 Single Time Step 205

10.3.2 Predominant Loop 207

10.4 Benchmark System Data Selection 207

10.5 Non-network Task Scheduling 209

10.5.1 Serial 211

10.5.2 Network and Non-Network Priority 211

10.6 Programming Considerations 213

10.7 Execution Performance 215

10.7.1 Serial Scheduling 215

10.7.2 Enhanced Scheduling Schemes 219

10.8 Observations and Conclusions 223

CHAPTER 11 CONCLUSIONS AND IDENTIFICATION OF FURTHER AREAS FOR RESEARCH 226

11.1 Conclusions 226

11.2 Suggestions for Further Research 230

REFERENCES 232

APPENDIX 1 - GLOSSARY 239

APPENDIX 2 - MULTIPROCESSOR PERFORMANCE MEASURES 242

APPENDIX 3 - SUBSTITUTION STEPS IN THE LU AND BIFACTORISATION METHODS OF SOLUTION OF LINEAR EQUATIONS 244

APPENDIX 4 - DEFINITION OF VECTORS USED IN ALGORITHM IMPLEMENTATION 247

APPENDIX 5 - CODED IMPLEMENTATION OF THE BGF PROGRAM SECTION 248

APPENDIX 6 - PAPER TO APPEAR IN 'IEEE TRANSACTIONS ON COMPUTERS' 250

List of Principal Symbols

Note that a glossary is provided in Appendix 1.

A - amperes

AC - alternating current

ASCII - American National Standard Code for Information Interchange

Baa - bus availability attenuation

bit - a binary digit

BS - bus switches

BSF - bus saturation factor

DC - direct current

EPROM - erasable programmable read only memory

FLOPS - floating point operations per second

IC - integrated circuit

I/O - input and/or output

M - memory units

MDS - microcomputer development system (especially INTEL)

MFLOPS - millions of floating point operations per second

μsec - microsecond

msec - millisecond

Nm - number of modelled processors

NMI - non-maskable interrupt

Nr - number of real processors

nsec - nanosecond

P - processors

PCI - programmable communications interface

PIT - programmable interval timer

PPI - programmable periferal interface

RAM - random access (read/write) memory

te - execution time

V - volts

VMS - virtual memory system (name of VAX operating system)

Abstract

Efficient multiprocessing approaches to the execution of digital computer programs, which analyse power system transient stability, have been investigated.

Different program sections are found to have greatly varying levels of effect on overall execution performance. The most demanding need which emerges is for a very efficient practical method to solve large sparse sets of linear equations. Without a satisfactory scheme, the maximum gain in performance over single processor operation is severely restricted. A suitable algorithm has been developed and is described.

To validate the effectiveness of proposed algorithms, practical multiprocessing hardware has been built. The equipment has also allowed evaluation of hardware requirements, in particular the capability of the inter-processor communication network. The parallel processor developed enables efficient program development and testing in an environment which is research oriented yet very closely resembles possible practical implementations.

The results of an execution simulation are combined with practical performance measurements to determine limits to the number of processors usefully employed, and the gains in performance over single processor operation achieveable.

When compared with other algorithms for solving linear equations, the one developed is shown to run very efficiently. To further improve performance, novel methods of mixing execution of the linear equation solutions and other sections of a transient stability analysis program have been practically implemented.

Acknowledgements

I wish to express my sincere thanks to my supervisor, Dr. C.P. Arnold, and to my co-supervisor Mr. M.B. Dewe, for their guidance and assistance throughout the course of this research.

Thanks are also due to Professor J. Arrillaga, Mr. J.G. Errington, and Mr. R. Harrington. Their experience and helpful advice has been invaluable. For their technical assistance, thanks go to Mr. A.R. Cox, Mr. M. LaHood, and the other technicians who were involved in often monotonous hardware construction.

The wholehearted cooperation of my post-graduate colleagues, especially Malcolm Barth, Jeff Graham, Hiroshi Hisha, and Jim Truesdale, is acknowledged. Thanks are also extended to the many students who have contributed through final year project work.

I am indebted to Mr. W.K. Kennedy and his hard working followers who maintain and develop, with little reward, facilities such as the computer used throughout this project and in the preparation of this thesis.

The assistance given by the University Grants Committee, both in supporting me through a Post-graduate Scholarship and in financing much of the hardware required for the project, is gratefully acknowledged.

CHAPTER 1

INTRODUCTION

Performance assessment is essential during both the development and operation of modern power systems. In most cases, and to an ever increasing extent, this assessment is done by simulating operation using digital computers. A variety of programs are used to analyse different facits of operation. Some are aimed at economic power dispatch, while others concentrate on maintaining security of supply.

Two conveniently discernable areas of power system modelling are instantaneous and dynamic simulation. An example of instantaneous modelling is provided by load flow studies, which are particularly useful in aiding economic power system operation. The processing capability of modern computers is not, in general, a constraint to the execution of load flow or other instantaneous modelling programs. Execution efficiency, on the other hand, becomes an important consideration in the design of dynamic simulation programs. Reasons for increased execution times include greater detail in component models used, and the increased number of executions needed to obtain a performance profile over a period of time.

Dynamic modelling programs can be categorised according to the range of time periods of interest. These periods correspond to, and are determined by, the time constants of both power system elements and disturbing phenomena. Figure 1.1, which originates from data presented by Concordia and Schulz (1975), illustrates various time range categories. Fast transients, due to lightning or switching surges for example, require power system models accounting for transmission line effects, such as reflection. Periods modelled within this program category range upwards

from fractions of microseconds. An example of the use of this form of analysis is in insulation co-ordination. For longer periods, over which mechanical changes of state can occur, transient stability analysis programs become applicable. These analyse the effect of large disturbances, such as faults, and require non-linear component models. As the time period under consideration is extended the detail of models increases, and there is good justification for using small perturbation frequency domain programs to reduce computational costs. For the purposes of the work presented in this thesis, transient stability studies involve consideration of periods in the order of a second. As such, much of the relatively slower responding equipment of power systems can be ignored.

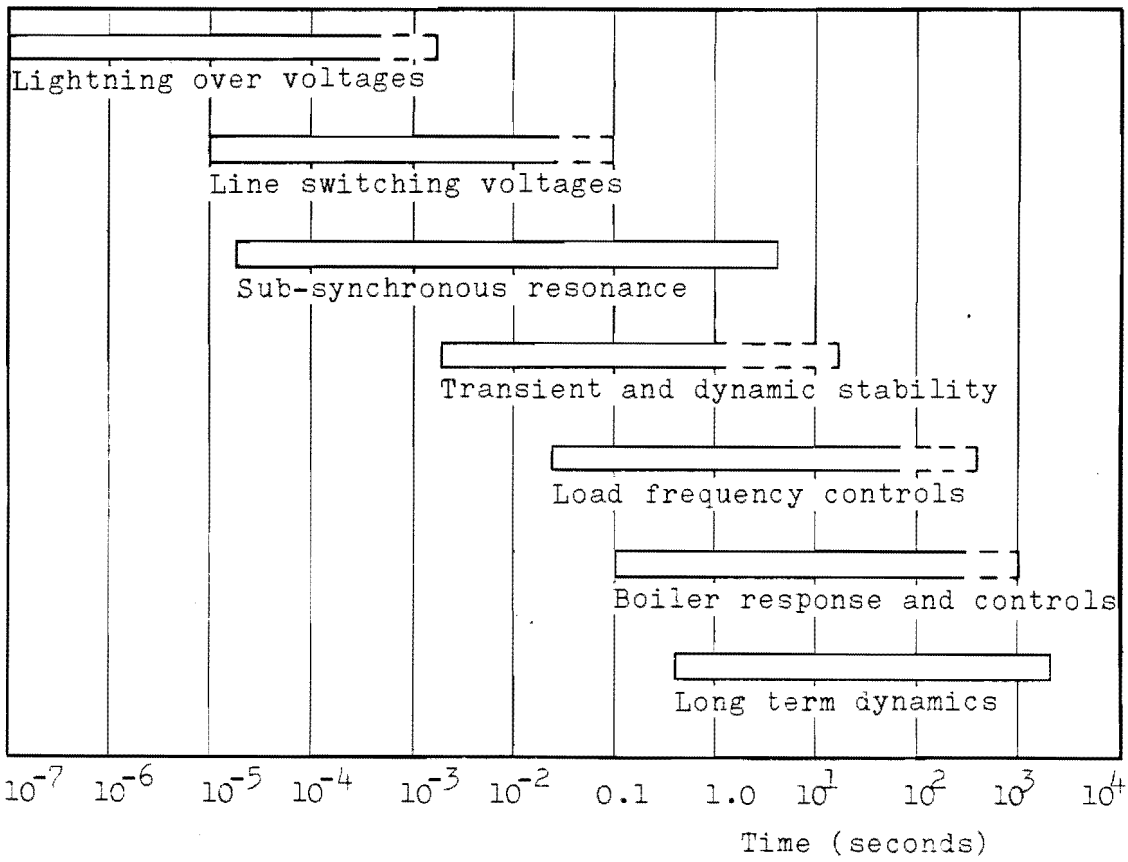


Figure 1.1: Time Range Categories for Power System Analysis

The possibility of transient instability must be considered in the

planning of systems and during operation. Adequate margins must be maintained to ensure continuity of supply to an acceptable portion of a system in all possible fault situations. Margins are affected by the instantaneous operating conditions of a network. Therefore, unless up to date information is available, worst case conditions must be assumed.

There is a current trend towards the use of on-line computer based power system control centres. This trend results from a continuing decline in computer costs coupled with increasing processing capability which enables more efficient and cost-effective use of power system distribution equipment. To continuously monitor the state of power systems, on-line data aquisition is required. This involves the use of transducers attached at points in the power system which relay information directly to the computers. In some cases the computer can also directly control aspects of power system operation.

Transient stability analysis is a computationally demanding task. Therefore, it is nearly always performed off-line at present and data is provided by an operator using punched cards or similar. This approach is quite satisfactory for system design studies and infrequent checks for stability.

As transient stability analysis produces results describing operation over a period of time, execution speed can be defined relative to real time. However, the complexity of component models, and the size of networks, significantly affects execution speed complicating accurate assessment of execution speed relative to real time. Some experience has been gained using a B6700 computer which has speed that is typical of available mainframe machines. Indications are that even very small systems cannot be simulated in real time. For instance, in the case of the smallest system tested, which has three buses and a single detailed

generator model, execution is 10 times slower than real time. For large systems this factor reaches the order of thousands.

Increases in the rate at which transient stability analysis programs are executed would be useful for a number of reasons:

.the cost of processing time is high,

.more frequent analysis would permit improved operating strategies, and

.real time analysis would open up a whole new set of possibilities for control and protection.

If faster than real time analysis was available, the effect of faults could be determined after they had occurred, and optimal strategies for isolating the problem could be implemented within critical clearing times. This form of protection is presently very much theoretical, as many problems, other than execution speed, must be overcome before it could conceivably be practical. For instance, the unreliability of input from transducers must be effectively handled. Approaches to the solution of this problem, (eg. by Brown, (1981)), could themselves be computationally demanding. In any case, protection of this form is inconceivable without radical improvements in computational speed.

The capabilities of computers are rarely well matched to the problem to which they are assigned. This difficulty is partly addressed by the availability of a wide range of computers with varying processing capability. Beyond differences in complexity, a match between computer and task can be achieved through two extremes in approach:

.Where a computer's capability exceeds the requirements of

any single function, many tasks can be handled concurrently by a single processor. This is called multiprogramming or multitasking.

.If a single processor cannot operate fast enough to implement a single function then many processors can be applied ie. parallel processing or multiprocessing.

Varying requirements for fast response and/or high throughput for different problem types are met by combinations of these possibilities.

This thesis describes developments aimed at increasing the execution speed of transient stability analysis programs through the use of multiprocessing. Over the past five years this field has attracted an increasing level of interest. In some cases it has been considered as a possible application of already developed hardware eg. using the CM multiprocessor (Dugan et al, 1979 and Durham et al, 1979). In others, the appropriate form of hardware is a design consideration.

The objective of multiprocessing, when applied to transient stability analysis, is to attain high execution speed. Priorities with respect to speed and to cost vary between execution environments. For instance, in off-line situations minimum execution cost will be important, while speed may be the overriding requirement, almost irrespective of cost, in on-line applications. Work presented in this thesis is directed towards the best possible utilisation of many processors. As such, to ensure cost-effective yet realistic research possibilities, consideration is directed strongly towards high processing efficiency rather than speed resulting from high individual processor execution rates. Approaches to the measurement of multiprocessor performance are described in Appendix 2.

Figure 1.2 illustrates the structure of this thesis with indication of the flow of information. In early Chapters (2 to 4), the problem is

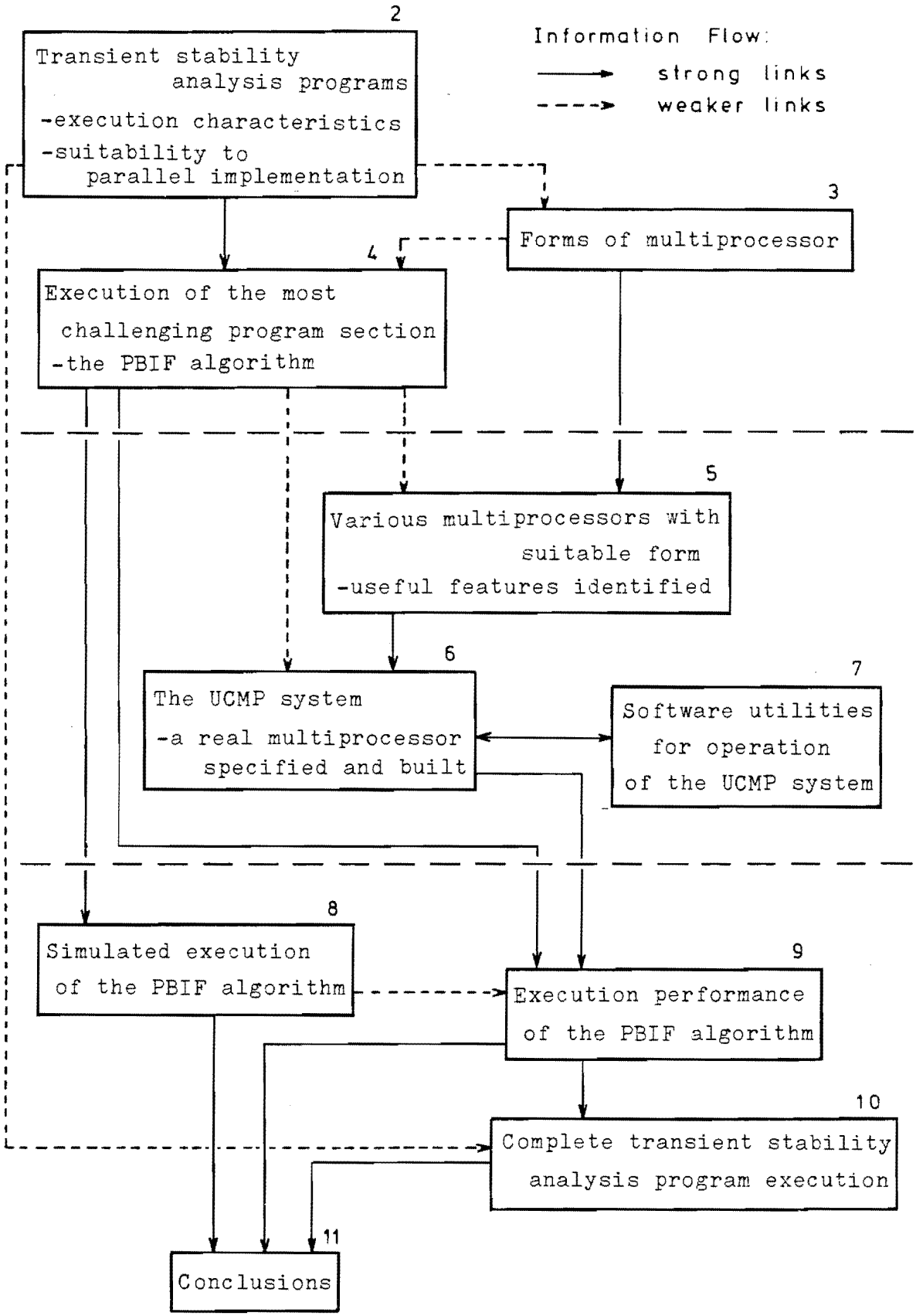


Figure 1.2: Thesis Structure

examined in detail and matched to an appropriate form of multiprocessor. A description of hardware, designed and implemented to realistically evaluate the performance of proposed algorithms, is then given in Chapters 5 to 7. Finally, in Chapters 8 to 10, both real and simulated implementations of algorithms aimed at high efficiency are described. A glossary, given in Appendix 1, provides separate definition of power system and computer related terms.

In Chapter 2 serially executed programs implementing transient stability analysis are outlined. Execution features relevant to possible parallel implementations are isolated, and a study reveals those areas most appropriately considered for multiprocessing. Of these areas, one is selected as likely to provide a serious restraint to execution speed ie. the solution of linear equations describing the interconnecting network.

Various forms of multiprocessor are introduced in Chapter 3. These are matched to the needs identified in Chapter 2 to determine the form most suited to transient stability analysis. An algorithm for efficient solution of linear equations is then developed in Chapter 4, aimed at implementation on the selected multiprocessor type. Many such algorithms have been described over the past few years, but an optimal practical approach has not yet been determined.

A number of existing multiprocessing systems are described in Chapter 5. Features considered useful are identified leading to the production of a real multiprocessor which is described in Chapter 6. Software enabling the operation of this system, in a number of modes, is outlined in Chapter 7.

For comparison with the real hardware's performance, and to provide

performance estimates for a wide range of execution conditions, a detailed simulation of the execution of the linear solution algorithm was developed. This is described in Chapter 8 along with a number of performance characteristics which indicate the efficiency of execution possible.

In Chapter 9, performance of the real hardware is merged with simulated characteristics to confidently ascertain the scope of the linear solution algorithm. Practical tuning of two program parameters is attempted. A problem, which only emerged during real operation, is described, and its level and methods of reducing its effect are discussed.

The linear equation solutions are combined with other program sections to estimate real hardware performance during execution of the complete transient stability analysis program in Chapter 10. A number of approaches, exploiting data independence, are implemented.

Many avenues for further investigation remain in this new field. It is clear when viewing the expanding, but already widespread, interest in the parallel execution of transient stability analysis programs, especially by research coordinating bodies such as the Electric Power Research Institute (EPRI), that developments in the area, if successful, will play a significant role in the future of power systems analysis.

CHAPTER 2

EXECUTION OF A TRANSIENT STABILITY ANALYSIS PROGRAM

2.1 Introduction

Synchronous machines have characteristics enabling stable parallel operation. Therefore, large A.C. power systems, centred around synchronous generators, are possible. The level of disturbances which would result in unstable operation is an important consideration during both design and operation. Transient stability analysis permits the assessment of stability limits and margins by simulation of power systems during disturbances.

In this chapter an introduction to the concepts of transient stability is followed by a description of digital computer based approaches to analysis. The execution characteristics of an analysing program are investigated with a view to parallel processing. A well proven program, based on a widely accepted solution scheme, is used. It is important to note that details of the modelling techniques employed are not important unless, in the context of parallel implementation, radical changes in the approach to solution are required.

Program sections which, when run, require a very high proportion of execution time are identified. These are selected as the most appropriate routines for detailed investigation with respect to parallel execution. From among the routines selected, one stands out as a likely restraint to efficiency.

2.2 Transient Stability Concepts

The object of transient stability analysis is to determine the ability of a power system to remain in synchronism after a disturbance. Transient instability is generally the result of a short duration, major change in network conditions eg. a fault. The period of interest in analysis is that between disturbance initiation and either regaining steady state stability, or complete loss of synchronism.

The concept of transient stability can be viewed in terms of a simple example. Consider the single synchronous machine shown in figure 2.1 which is generating power fed into an infinite system through a purely reactive impedance, X.

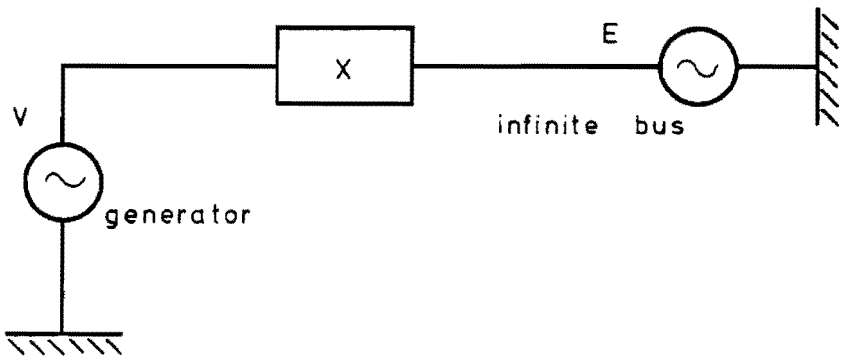


Figure 2.1: Single Machine Power System Example

Provided any saliency of the generator is ignored, the electrical power, Pe, transferred to the infinite system can be expressed as:

$$Pe = \frac{|V|\,|E|}{X} \sin \delta \qquad (2.1)$$

where δ is the phase angle between voltages V and E, which is strongly related to the mechanical rotor position within a synchronous rotating frame of reference

2.2.1 Steady State Stability

With all other factors held constant, the variation of Pe with δ is illustrated in figure 2.2. Changes in requirements for delivered power are met through changes corresponding to alteration of the synchronous position of the machine rotor. The point at which electrical power output begins to decline with further increase in δ is called the steady state stability limit ie. where Pmax is delivered at angle δ_{sssl}. Attempts to slowly increase power transfer, for instance, by increasing mechanical power input, which force δ to exceed 90^0 result in loss of synchronism. This is because the electrical power delivered cannot match the mechanical power input resulting in a power surplus which adds inertia to the rotor ie. changing its speed.

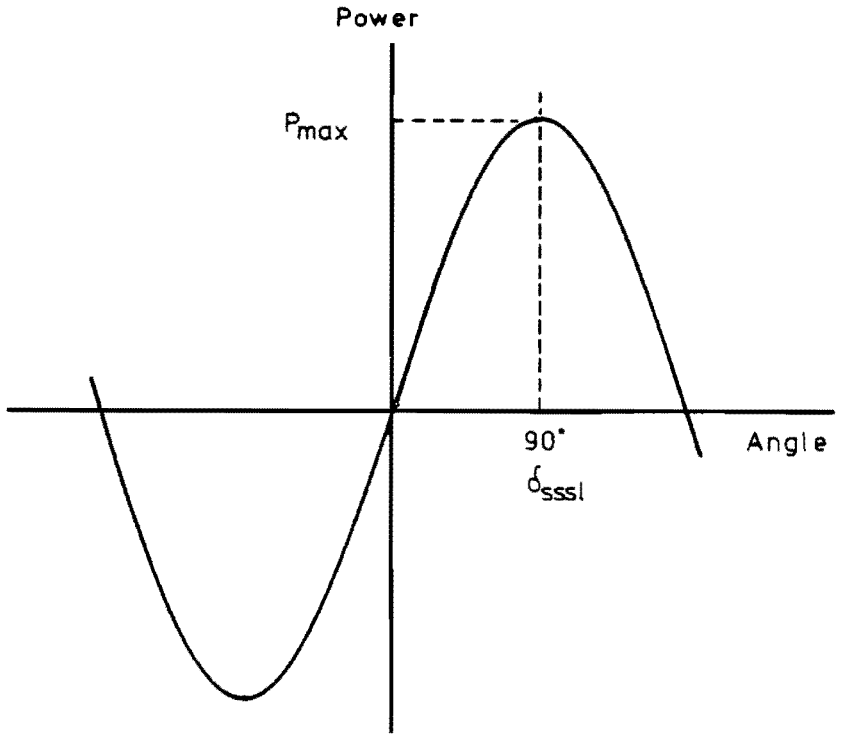


Figure 2.2: Relationship between Real Power Delivered and Synchronous Machine Rotor Angle

2.2.2 Transient Stability

Rapid changes in input power are not accompanied by instantaneous repositioning of rotor angles and δ. Assume the power delivered by the generator is to be increased. Mechanical input is increased appropriately. The consequent instantaneous difference between input and output power is taken up by acceleration of the rotor. As shown in figure 2.3, once the new steady state angle δ_2 is reached, the rotor decelerates rather than immediately assuming synchronous speed. Hence, overshoot to δ_3 occurs and, provided there is damping in the system, the rotor eventually settles to synchronous operation at angle δ_2.

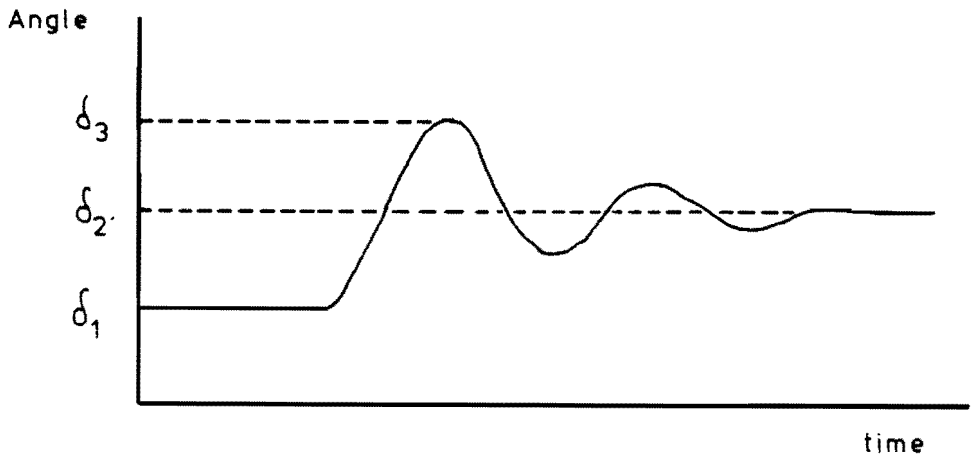


Figure 2.3: Rotor Angle Variation During the Transient Period

During such transients, rotor angles can exceed the steady state stability limit and stability can be maintained provided that, during these times, the electrical power does not fall below the mechanical power. The maximum change in power, and the maximum value of δ can be determined using the P-δ curve. To maintain stability the nett rotor speed change must be zero. To achieve this, the accelerating and decelerating energy must be equal ie. the integral of accelerating power* over time must be zero. It can be shown (eg. Byerly and Kimbark, 1974) that this condition can be

restated as: the accelerating power integrated over the angle δ must be zero.

In terms of a P-δ diagram, figure 2.4, a change from P_1 to P_2 will result in angles up to δ_3 which can be determined by equating accelerating and decelerating areas.

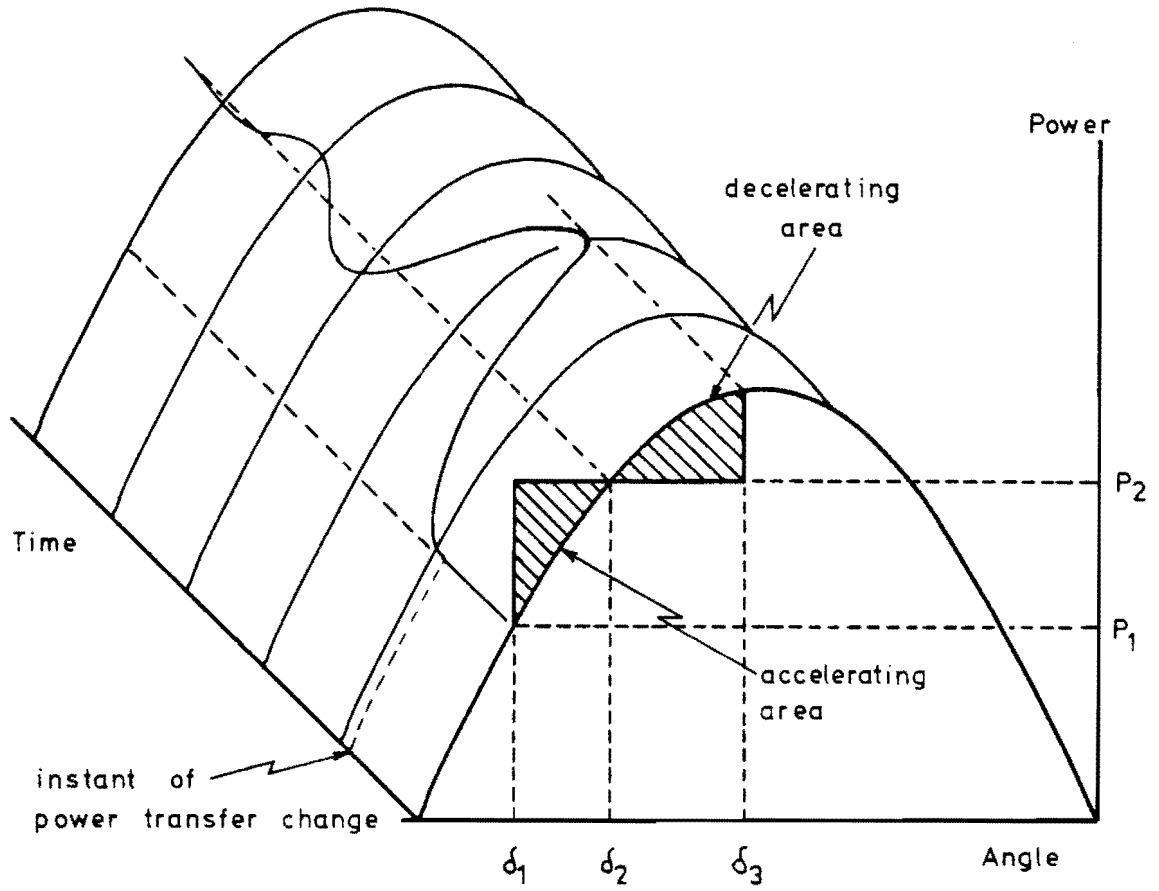


Figure 2.4: Rotor Angle Variation in Time Related to the P-δ Curve

The limiting case for stability is illustrated in figure 2.5. After passing the steady state stability limit, the accelerating power is still

---

* - accelerating power is the difference between mechanical input and electrical output powers.

negative until the point δ_{tsl} is reached. This point, the transient stability limit, will just be reached if $A_1 = A_2$. Comparison of A_1 and A_2 to determine stability is referred to as use of the 'equal area criterion'.

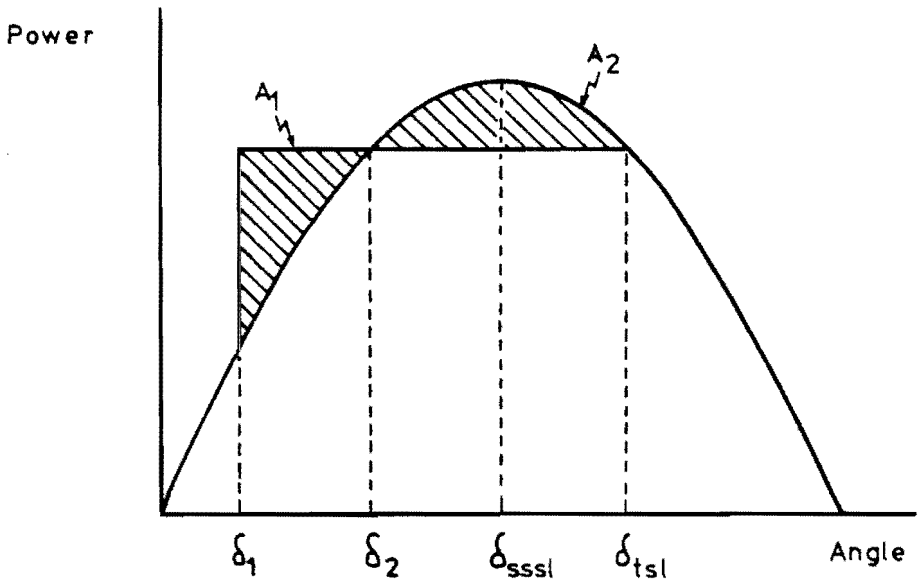


Figure 2.5: Equal Area Determination of the Transient Stability Limit

These principles can be extended to the case of fault conditions where the situation can be modelled as a sequence of changes from one P-δ curve to another. Consider the case depicted in figure 2.6. The three curves represent the pre-fault, fault, and post-fault conditions of a network. The transition from fault to post-fault periods occurs, for instance, at the opening of circuit-breakers. The accelerating and decelerating areas are shown in the figure. Stability is reached at stable operating point 6, where there is no change from the original power delivered. The transient period can be viewed as a sequence of transitions, in ascending order through the numbered points, culmininating in damped oscillations about point 6. The rate at which the rotor angle synchronous position moves is dependent on the rotor's inertia. Taking account of the inertia, the maximum time before the circuit breakers must

operate can be determined by equating accelerating and decelerating areas at the stability limit. Establishing critical clearing times is an important application of transient stability analysis.

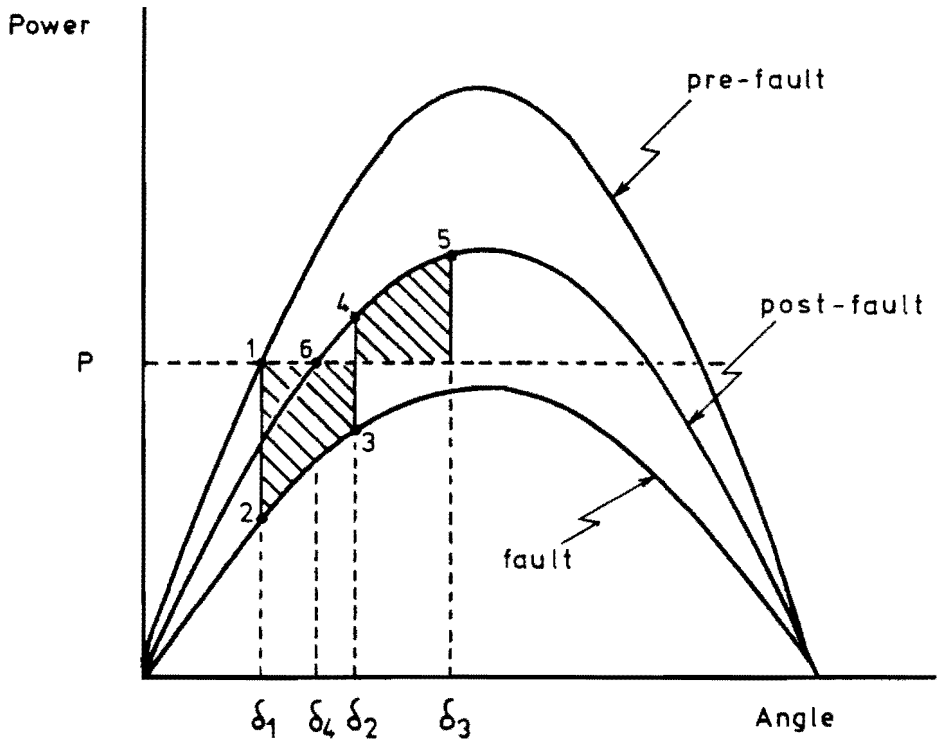


Figure 2.6: The Equal Area Criterion Applied During Fault Conditions

2.3 Transient Stability Analysis

The equal area criterion is useful to illustrate the principles of transient stability, and for the analysis of small systems. Its applicability is limited, however, for practical multi-machine systems.

Modern transient stability analysis is restricted almost entirely to digital computer based implementations. The work presented in this thesis is centred on a computationally efficient program used in the Dept. of Electrical Engineering at the University of Canterbury (Arnold, 1976). It will be referred to as the UCTS program.

2.3.1 Approaches to Solution

Following determination of initial conditions using steady-state load flow techniques, a power system can be modelled by two sets of equations. To ascertain system performance, these equations must be solved simultaneously over a period of time. The first set is algebraic

$$g(X,Y) = 0 \qquad (2.2)$$

The second set is differential

$$dY/dt = f(X,Y,t) \qquad (2.3)$$

where:

g is a set of functions describing the steady state relationships in a network. This includes network, steady state load models and algebraic machine equations. Justification for the use of steady-state models is provided by the fact that the system responses involved are much faster than those of synchronous machines, and, therefore, can be assumed to be instantaneous.

f is a set of functions describing the dynamic behaviour of machines and their controls.

X are the non-integrable variables which depend on a set of algebraic constraints. Examples include the voltages and currents at network nodes.

Y are integrable variables. Their values cannot be assumed to change instantaneously as can the values of X. The generator rotor angles and internal voltages are examples.

Methods for solving these equations fall into two categories. One category includes approaches which estimate performance in a single solution step. Implementations of these direct methods, using Liapunov

functions, can be executed very quickly and, hence, offer the opportunity for real-time application with on-line process computers (El Guindi and Mansour, 1982). The methods, however, are prone to large errors and are useful only to provide rough estimates. The second category involves methods which evaluate system state at each of a number of time steps.

The solution method applied in the UCTS program falls into the second category, and is based on the approach described by H.W. Dommel and N. Sato (1972). The differential equations, (eqn. 2.3), forming part of the generator models, are transformed to new equations to enable implicit trapezoidal integration. These 'trapezoidal' equations are combined with the algebraic generator modelling equations, (part of eqn. 2.2), leaving the linear network equations separate. The two sets of equations formed, one describing generators and the other the network, can be solved by iteration. Convergence is improved by including appropriate generator admittance terms in the network model. Non-linear, non-generator elements can be included in solution in a manner similar to that used for the generators. Therefore, throughout this thesis, all non-network component models are included under the heading 'generator'. An alternative approach to the solution, which can be implemented after the formation of the trapezoidal equations, is to use Newton-Raphson techniques linearising the generator models. Execution is slowed by the need to recalculate values of elements within the Jacobian many times. When compared, the direct method seems preferable to Newton-Raphson based schemes (Dommel and Sato, 1972).

The linear equations describing the network can be combined in a matrix representation:

$$[I] = [Y][V] \qquad (2.4)$$

where [V] is unknown.

[Y] is generally very sparse with no more than a few percent of elements being non-zero. Solution of equation 2.4 could be achieved by explicit calculation of $[Y]^{-1}$. This process, however, does not preserve sparsity and results in unnecessary demands for both storage and computation time. A very efficient algorithm has been developed for the direct solution of sparse networks without the formation of $[Y]^{-1}$ (Zollenkopf, 1970). This method, called bifactorisation, is based on Gaussian ordered elimination and preserves much of the sparsity of [Y] and processes only non-zero elements. Algorithms based on factorisation are more simply described in terms of the computationally similar LU substitution approach to linear equation solution. Consequently, LU substitution, which is outlined in Appendix 3, is employed in description throughout this thesis. After the formation of factor matrices, solution involves forward and backward substitution steps.

2.4 Serial Execution Characteristics

Several characteristics, identifiable in serial execution, are valuable in specifying an effective approach to parallel implementation of a problem solution scheme. Relevant characteristics determined in this section are:

- .the computational blocks which exist in the UCTS program,
- .the execution time of each block, and
- .the dependence each block has on its predecessors.

Of these, the third characteristic is very important in determining the efficiency that will be achieved in a multiprocessor. The level of dependence and an associated property, parallelism, are not easily determined using conventional program descriptions eg. flow-charts and

structure diagrams.

2.4.1 Parallelism and Dependence

Parallelism is a property describing the extent to which an implemented program can be distributed among a number of processors. A high level of parallelism is a fundamental objective in the development of parallel programs.

A task can only be executed when results from its predecessors are available. The number of tasks which can be executed at an instant is strongly related to the dependence of tasks on their predecessors. Low levels of dependence correspond to a high degree of parallelism and should, therefore, be identified in creating efficient parallel algorithms.

The following example serves to illustrate these concepts, and to introduce tasks graphs as an effective development tool:

Determine f(x), given x where :-

$$f(x) = (x^2 + x^3)/(2x^2)$$

An approach to the solution is illustrated in figure 2.7. The following terms, used by Arnold et al (1983), are necessary in describing the representation employed.

Task graph: a directed graph depicting the order of execution and times of synchronisation during the running of program code ie. data flow

Nodes,edges: the components of a task graph. Nodes correspond to operations in the solution process, and edges depict the dependence of operations on the results of previous operations.

In,out-edges: the edges entering and leaving a particular node, respectively

Process: a block of program code executed serially by a single processor

Task: the program code associated with a node

Synchronisation: this is necessary when a processor requires the results of one or more processes executed in other processors to continue its own execution

Predecessors: nodes whose out-edges are in-edges of the node considered

States can vary during execution. Any that do are referred to as dynamic. At any time a task will have one of four states :-

not ready: some necessary preceding results are not available so the task cannot be executed

ready: all necessary results for initialisation are available and the task can begin execution

running: the task is presently being executed

completed: execution is completed and hence all all results produced by the task are available

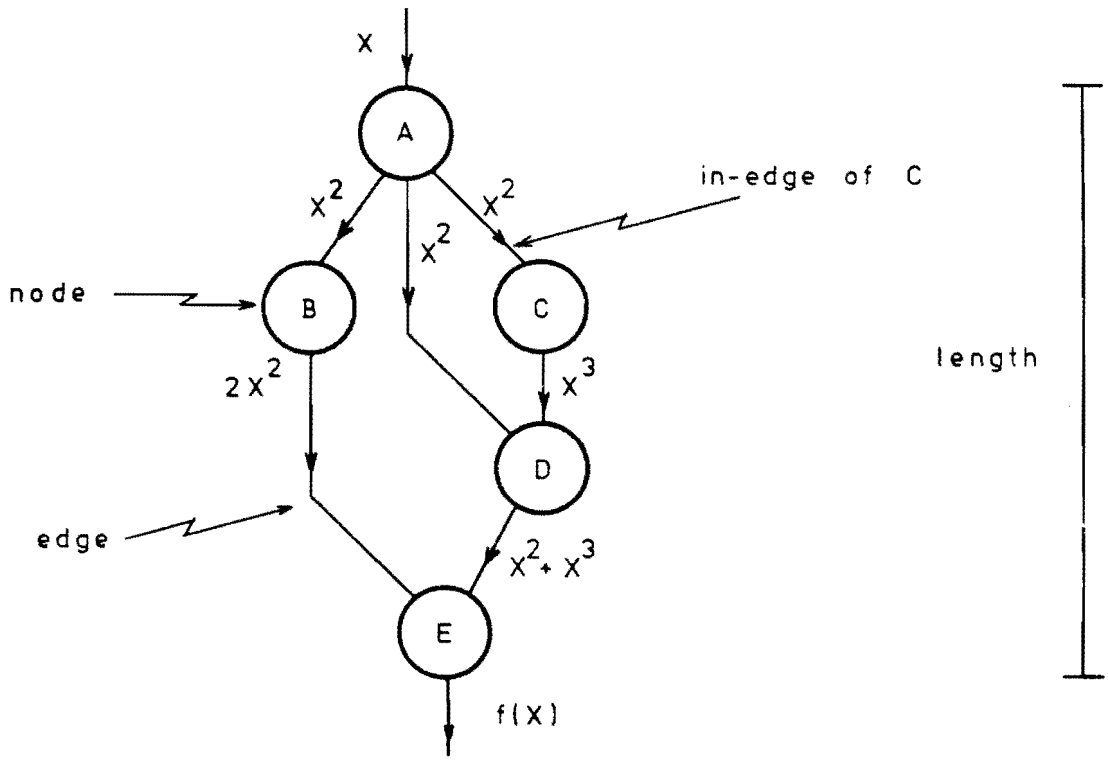


Figure 2.7: Task Graph Depicting the Steps Required in Determining f(x)

Dependence is observable in the task graph. Task E, for instance, will only be ready once both tasks B and D are completed. Tasks B and C illustrate parallelism as both have task A as their only predecessor. Once task A is completed they could be executed in parallel. Minimal multiprocessing execution times can be estimated using the length of the task graph ie. the number of nodes traversed in the longest path through the graph (four in the example). This approach, assuming unit execution time, is frequently used in assessment of the performance possibilities of an algorithm. Parallelism is strongly related to the width of the task graph.

2.4.2 Computational Blocks

The sequence of occurrence of execution steps for the UCTS progam is illustrated in figure 2.8. The outer loop involves determination of the power system state at each of many time steps. Within each time step changes of network are accounted for, and preparation is made for integration. This is followed by an iterative sequence converging to an estimate of system state at the end of the time step. Within the iterative loop data is exchanged back and forth between network and generator related processes. As each generator is modelled in a local frame of reference, translation to and from the network frame of reference is necessary.

2.4.3 Categorisation of Blocks

The program sections in figure 2.8 are separated into groups which can be usefully investigated with respect to the distribution of execution time. Such a study has been made to identify the program section categories which would most usefully be quickly executed ie. those requiring the majority of serial execution time. The selection of categories is based on the experience of other researchers, eg. Brasch et al (1978), and on dependence considerations discussed later. The three categories used are:

N - network related substitutions in the central iterative loop

G - generator related processes within time step solutions ie. including the iterative loop; and axis translations

O - other processing

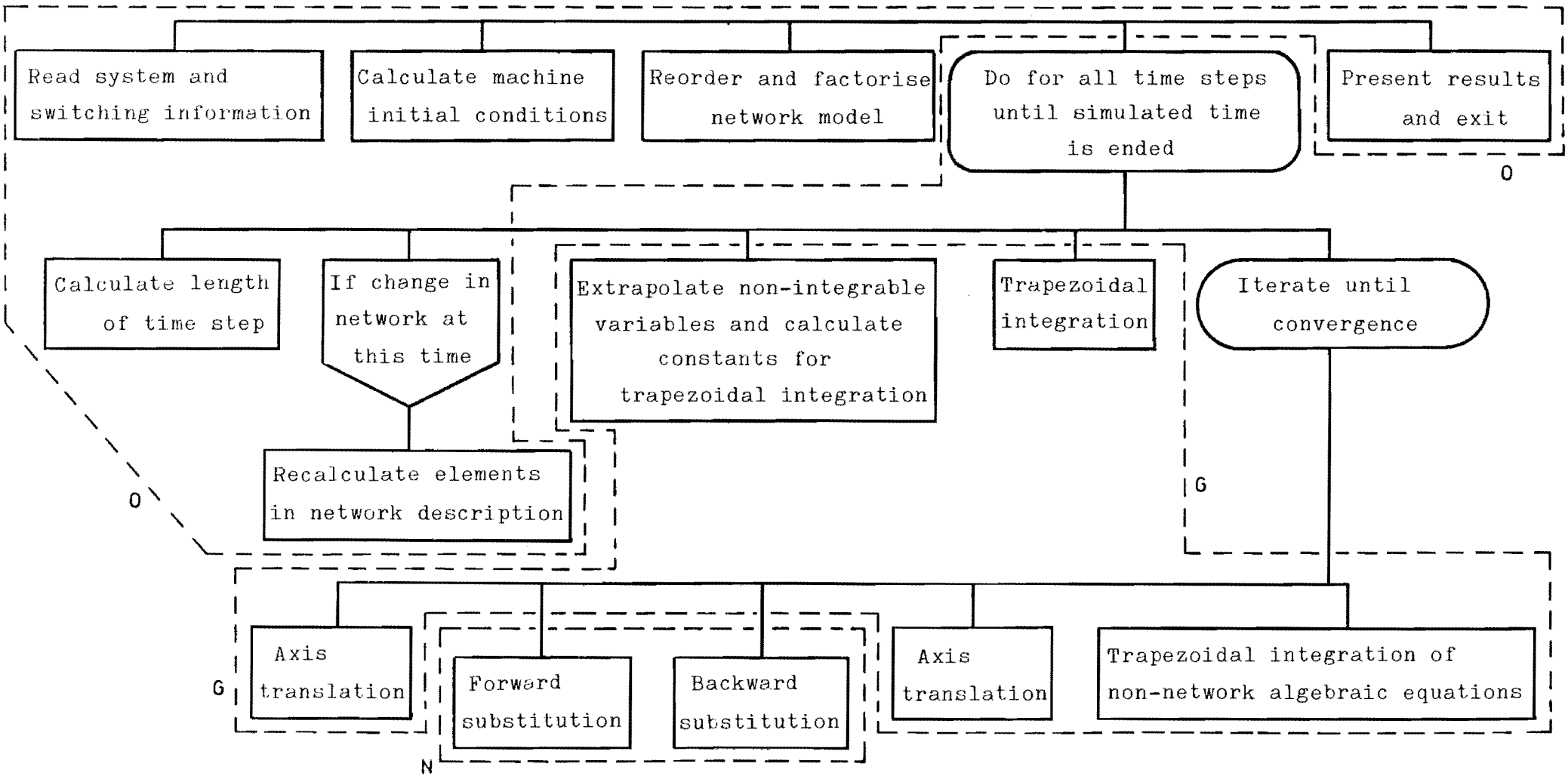


Figure 2.8: Execution Sequence of the UCTS Program

2.4.4 Distribution of Processing Time

Three distinct processing environments were used in obtaining the sets of processing time distribution information presented:

.a CDC-6600 computer used by F.M.Brasch et al (1978). They used the BPA (Bonneville Power Administration) transient stability analysis program which is structurally similar to the UCTS program.

.a B6700 computer executing the UCTS program

.an 8086 microprocessor based system using the program described in Chapter 10. In this implementation only the N and G process categories were considered.

Small timing errors were introduced in the B6700 based execution due to its use of virtual memory ie. some paging time is occasionally included. However, perusal of many detailed timing measurements indicated that the overall effect was insignificant.

A wide range of practical power system models were used as examples. Table 2.1 summarises the form of the systems employed. In the CDC-6600 investigations large networks were employed while small ones were used with the B6700. The same small systems were included among a greater variety tested using the 8086. As will be seen, the consistency of results ensures that the number of systems modelled is more than adequate.

Both the measured distribution of processing times and an indication of total execution times for all examples are depicted in figure 2.9. The three process type categories are identified using the letters defined previously. Execution times are shown relative to other examples using the same processor.

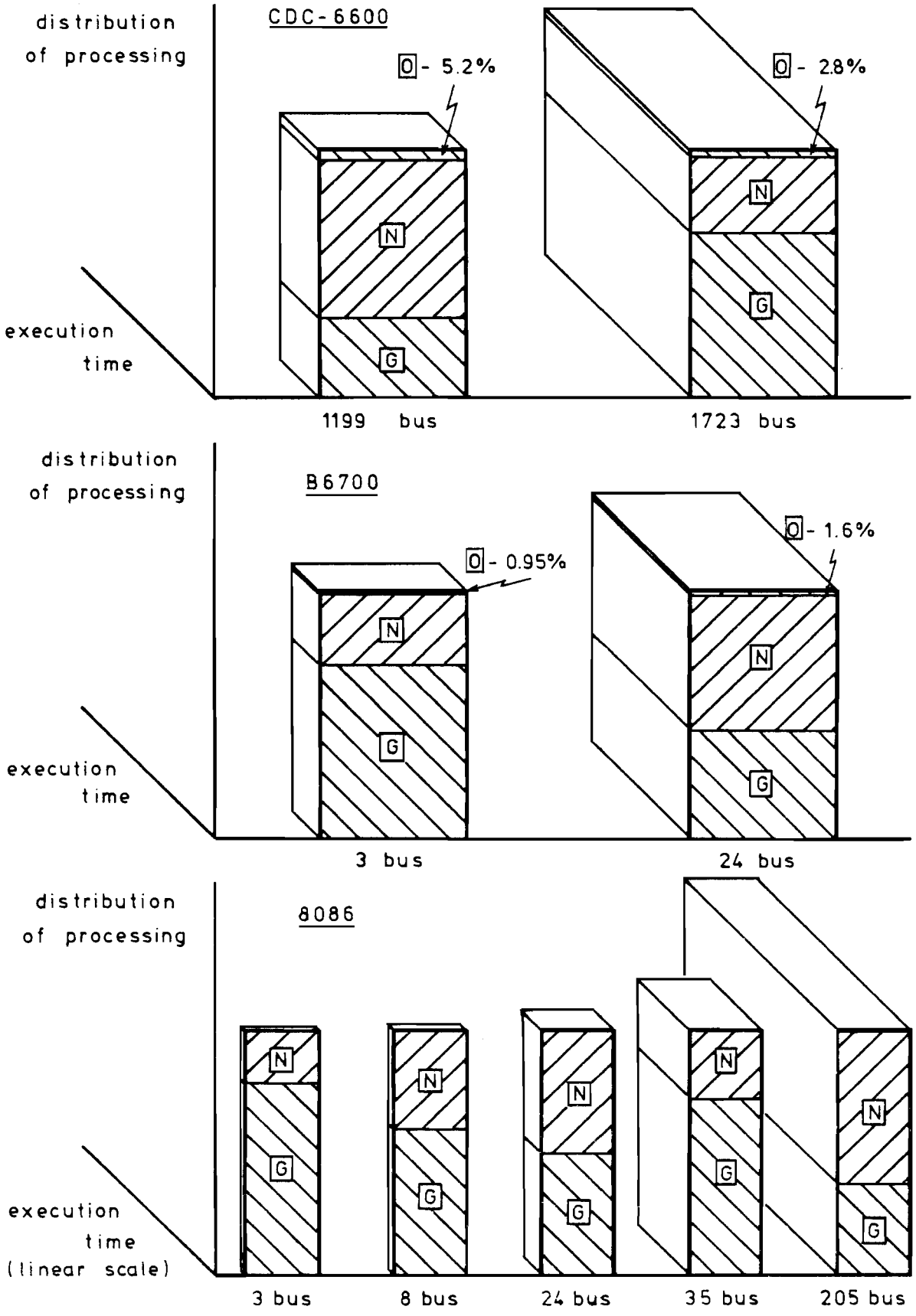


Figure 2.9: Distribution of Processing Time for Three Processor Types (with indication of the total execution time)

System Name	Number of Generators	Number of Buses	Sparsity Coefficient
1723-bus	398	1723	-
1199-bus	230	1199	-
3-bus	2	3	22.2%
8-bus	2	8	65.6%
24-bus	5	24	87.5%
35-bus	24	35	93.75%
205-bus	30	205	98.1%

Table 2.1: Networks Used in Execution Timing Tests

In the B6700 and CDC-6600 implementations the number of time steps and the number of iterations per time step are assumed to be indicative of typical execution situations. For the 8086 a single time step sequence is investigated with the number of iterations fixed at three, a figure which is typical during practical executions. (Note that the 8086 implementation is discussed in Chapter 10.)

2.4.5 Identification of the Area Most Suited to Parallel Execution

From the distribution of processing time shown in figure 2.9 two key points are noted:

.execution of generator and network related routines within the time step loop dominates the use of processing time in all cases, and

.portions of execution time of similar order are used by the generator and network related routines.

It is clear, therefore, that the program sections most usefully considered for performance improvement are the network and generator related routines within the time step loop. This is not to say, however, that other sections could not be efficiently implemented on a parallel processor. The parallel implementation of the formation of triangularised matrices, for instance, has been considered by J.W. Huang and O. Wing (1978).

Although they require a dominant proportion of processing time, the processes within the time step loop form a manageable proportion of the UCTS program's source code. It is likely, therefore, that, in terms of effort required, the parallel implementation of these sections would be most cost effective.

Work presented in this thesis concentrates on execution within the time step loop. To achieve high efficiency, both the network and generator related routines must be carefully considered with regard to scope for parallel implementation. In the following section a qualitative analysis of the dependence properties of both identifies the one most likely present difficulties during concurrent execution.

2.5 Parallel Execution Within the Central Loop

In power systems the network interconnects all of the buses. Generators are situated at individual buses and have no effect on each other except through the network. These physical dependencies are reflected in the coded implementation of power system component models.

2.5.1 Generator Models

Each generator is represented by a sequence of code which, when executed, translates a network nodal voltage estimate to a new estimate of

current injected at that node. If all of the network voltages are available, this independence provides the opportunity for simultaneous execution of all the generator models. A data flow representation of this execution possibility is given in figure 2.10.

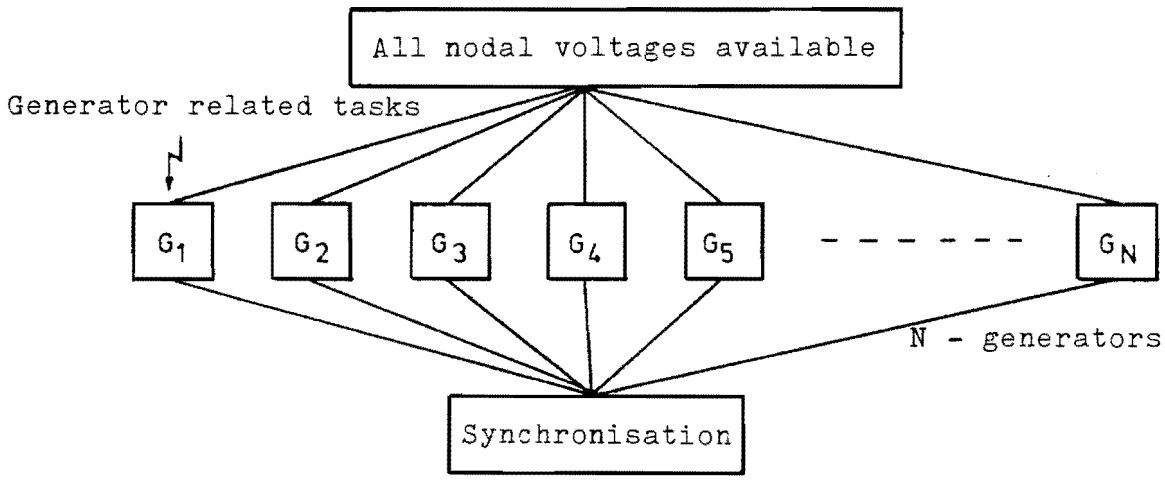


Figure 2.10: Task Graph Illustrating the Independence of Non-Network Related Tasks

Unless the number of processors is in the order of, or greater than, the number of generator models, efficient distribution of tasks among processors will be simple.

2.5.2 Network Models

Given the currents injected at every node, network models execute to produce corresponding voltage estimates. Current injected at any bus in a network can affect the voltage at any other. The matrices describing networks, although sparse, contain many elements representing these dependencies. Tasks cannot be defined, therefore, which make all the calculations associated with a particular node without consideration of the states at other nodes.

2.5.3 A Bottleneck

The effect of a bottleneck may be considered by way of an analogy. A vehicle travels two kilometers. For the first kilometer its speed is 100 KPH while for the second it is 1 KPH. Resulting average speed is a little under 2 KPH. The second kilometer, which constitutes a bottleneck, severely restricts average speed. Note that average speed is far more sensitive to the rate during the bottleneck kilometer and that improvement in average speed would be most effectively achieved through increases in speed during the bottleneck.

Because of their dependent nature, network models are likely to form a bottleneck in parallel execution. Overall performance will, therefore, be very sensitive to the approach used in network related execution. Consequently, effort aimed at improving the solution of linear equations would be valuable.

2.6 Summary

From serial execution timing measurements and qualitative identification of problem areas, two conclusions are derived directing the approach taken in the following chapters:

- the generator and network related routines in the central program loop are to be implemented in parallel, and
- the network routines, implementing the solution of large sparse linear matrix equations, are likely to present most difficulty.

CHAPTER 3

MULTIPROCESSOR TYPES AND THEIR SUITABILITY TO TRANSIENT STABILITY ANALYSIS

3.1 Introduction

Thousandfold increases in computer performance occurred between 1944 and 1951 and again between 1951 and 1964 (Enslow, 1974, p.3). Unfortunately, this rate of increase in speed has not and cannot be continued. The limiting factor has become the speed at which electrical signals travel. Extreme miniaturisation is a possible means of further speed enhancement, but this too has limits. Hence, since the 1960's there has been a great deal of interest in the use of simultaneous processor operation, ie. parallel processing, to increase computational throughput.

Another major area to which parallel processing has been applied is that of reliability and availability improvement through the use of redundancy of processing elements. A combination of increased throughput and enhanced reliability may well be useful in power systems applications eg. for on-line analysis and control. In this thesis, however, consideration is limited to speed improvement.

Many forms of parallel processor exist. They vary in cost, complexity of construction, complexity of operation, and suitability to specific problem types. In some cases the principles of operation of computational elements are being re-examined in the light of parallel processing requirements. In this chapter background descriptions of various families of multiprocessor are presented. Following this, consideration of the suitability of these families leads to a decision

regarding the type most suited to transient stability analysis.

3.2 Classification of Multiprocessors

The hardware defined way in which processors operate within a computer system can be called the model of computation. By far the dominant model of computation currently employed is the von Neumann approach. Two features which characterise the von Neumann model (Agerwala and Arvind, 1982) are:

.a globally accessible memory which can contain both program code and dynamically alterable data, and

.a program counter ie. a register whose contents define the address at which the current or next executable instruction would be found.

These features imply an ordered sequence of execution and sequentially accessible memory.

The majority of multiprocessors can be viewed as groups of von Neumann machines. However, special problems, which arise in parallel program implementation, have led to consideration of and, in a few cases, implementation of alternative models. All but the last processor type discussed in this section are based on von Neumann structures.

3.2.1 SISD - Serial Processors

Serial processors can be described as Single Instruction Single Data stream (SISD) processors. However, if pipelining, a feature discussed later, is used, SISD processors can be considered to be multiprocessors. The method of operation of SISD processors, illustrated in figure 3.1, is to take one instruction and produce one result per execution cycle. Most commercially available computers, from microprocessors to large mainframes,

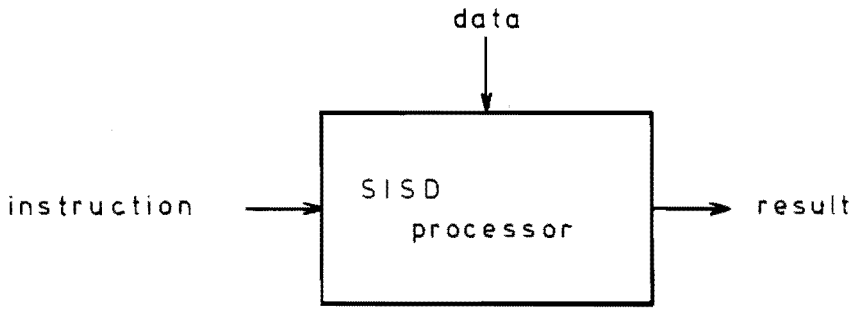


Figure 3.1: Instantaneous Operation of an SISD Processor

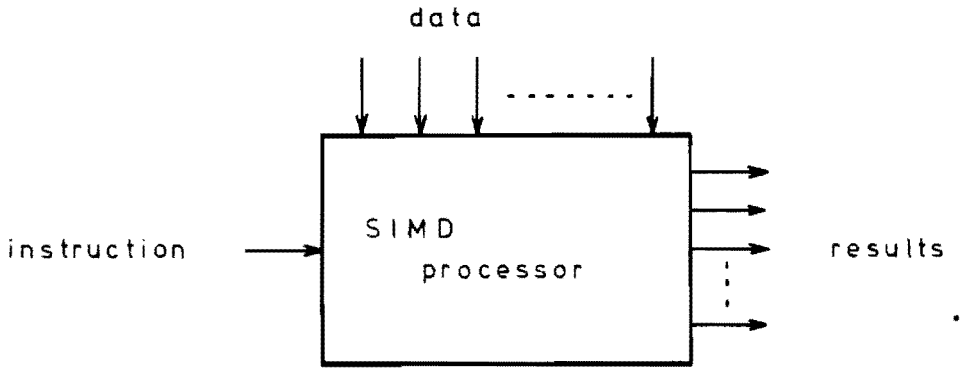


Figure 3.2: Instantaneous Operation of an SIMD Processor

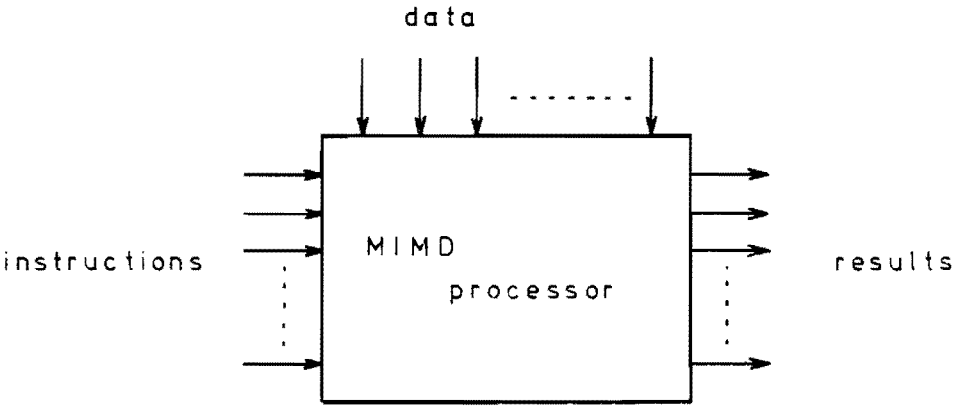


Figure 3.3: Instantaneous Operation of an MIMD Processor

fall into the SISD category.

Software for SISD systems is well established. Programmers are accustomed to using serial translators, eg. compilers, to generate serial code. Graphical aids to logical, effective serial program development, such as structure diagrams and flow charts, are widely taught and used.

3.2.2 SIMD Processors

Single Instruction Multiple Data stream (SIMD) processors contain a number of processing elements which operate simultaneously on separate data. As illustrated in figure 3.2, all processors execute the same instruction. SIMD processing systems are often referred to as vector or array processors. This terminology, however, is not universally accepted eg. the expression 'vector processor' is used by Happ et al (1979) to refer to all parallel processors.

SIMD processors are best suited to exploitation of parallelism at the instruction level ie. if many elements of data undergo the same operation. The obvious example of this situation is in the manipulation of vectors. The efficiency of processor utilisation is determined by how often the processing elements can be presented with full vector operations. Poor performance can result from either of two main problems:

.a predominance of scalar operations, and

.sparsity in vectors

Software requirements for SIMD systems are more complex than for SISD. Assuming user transparency is desired, (ie. the programmer sees only an increase in performance rather than an increase in programming difficulty as well), translators must search for suitable vector instructions and organise the distribution of program sections among the

processing elements. Many commercial array processors are sold with software providing user transparency.

More than any other form, currently available supercomputers obtain high throughput by using SIMD processing. The CRAY-1 (Russel, 1978) for example, uses an SIMD pipelined architecture and has a maximum execution rate of 140 MFLOPS (millions of floating point operations per second).

SIMD systems often operate as slaves to host computers. In this way the performance of existing installations can be enhanced without replacement. Examples from a multitude of available systems include (Computer Advts, 1981): the Applied Dynamics AD-10 which is hosted by DEC PDP-11's, the Floating Point Systems AP-120B which can be hosted by a number of computers, and the Sky Computers MNK-02 which can be hosted by either a DEC LSI-11 or an INTEL 8085. The price of array processors tends to be low if measured in terms of dollars per maximum operating speed eg. the MNK-02 is advertised as costing a few thousand dollars for MFLOPS performance.

3.2.3 MIMD Processors

As shown in figure 3.3, Multiple Instruction Multiple Data stream processors execute independent instructions on separate data simultaneously. Parallelism between tasks of any size can be exploited. As such, the range of problems which can be efficiently executed in parallel is expanded in comparison with SIMD implementation. On the other hand, the cost and complexity of MIMD systems is significantly greater than SIMD systems with similar maximum throughput. A trade-off, therefore, emerges between the cost of implementation and the degree of exploitation of parallelism. Because of the cost difference, MIMD implementation is likely to only suit problems which are very poorly handled by SIMD

approaches.

Effective software development for MIMD computers is difficult. Removal of the SIMD constraint forcing the utilisation of instruction level parallelism only introduces a complex choice of a suitable or optimal level at which to divide a problem into tasks. Before even considering the distribution of tasks among processors, identification of all of the parallelism available within a problem is not simple. Having established the opportunities for exploitation of parallelism, factors influencing selection of distribution scheme include:

.the number of processors,

.the complexity of programs required for implementation,

.the management overheads introduced, and

.the overheads arising through sharing of hardware resources.

Some recently developed languages (eg. Ada and concurrent Pascal) include constructs which allow the programmer to specify tasks which may be executed simultaneously. However, a gap remains as no systematic, computer implementable, method is currently available to enable optimal selection and distribution of tasks among processors. Therefore, efficient code production cannot be transparent to the user.

Due to both hardware and software complexities, MIMD systems are commercially far less common than SIMD machines.

3.2.4 Pipelined Processors

Pipelining can be viewed as a special case of MIMD processing. As illustrated in figure 3.4, all but one of the resulting data elements are

fed back for further processing at each execution step. In practice, as well as this restriction to data path, instructions are fixed. The function of pipeling is, perhaps, more meaningfully depicted in figure 3.5. The origin of the name, through analogy with fluid flow through a pipe, is clearer in this diagram. Instructions enter the pipe, travel through, and are completed on leaving the pipe. Speed enhancement occurs as a number of separate instructions can be in the pipe simultaneously.

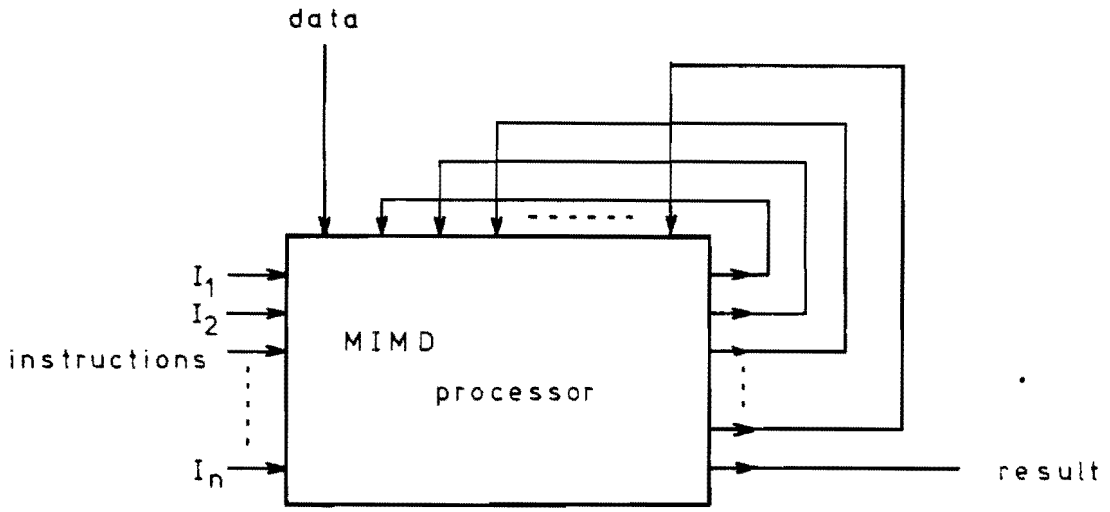


Figure 3.4: MIMD Processor Configured for Pipelined Operation

Figure 3.5: Operation of a Pipelined Processor

A simple and commonly implemented example of pipelining is in the division of basic instruction executions into separately executable tasks ie. :

(a) the fetching of instructions from memory,

(b) the decoding to establish the operation required, and

(c) the execution of the operation.

Another common use of pipelining, more restricted to larger machines, is in the implementation of complex instructions involving a fixed flow of information eg. floating point operations.

Given an MIMD system, pipelining could be employed as a mode of operation. However, there seems little point in restricting MIMD operation in this way. Pipelining is more likely to be employed either in SISD machines, or to augment the parallel execution capabilities of SIMD or MIMD systems. Very few modern processors do not utilise pipelining to some extent. At a simple level it is even used in microprocessors eg. the 8086 has a separate instruction pre-fetching facility, the BIU or bus interface unit (INTEL(g), 1979).

When pipelining is used, not only is the parallel nature of executions transparent to the user, but it is also transparent to the software. This has probably been a motivating factor in its extensive application.

3.2.5 Data Flow Processors

As introduced in Chapter 2, data flow descriptions of program execution sequence are very useful in determining task dependence. A convenient method of parallel program production would be a direct mapping from data flow diagram to computer language. Such languages have been developed eg. Val and Id (Ackerman, 1982). Program code written in one of these languages could be translated to a form suited to execution on a conventional computer. However, a more appropriate hardware configuration implies a radical change from the von Neumann approach.

A conceptual data flow processor is depicted in figure 3.6. Elements within the processor observe the availability of data and determine consequently enabled instructions. Execution units, if available, are then assigned to run these instructions.

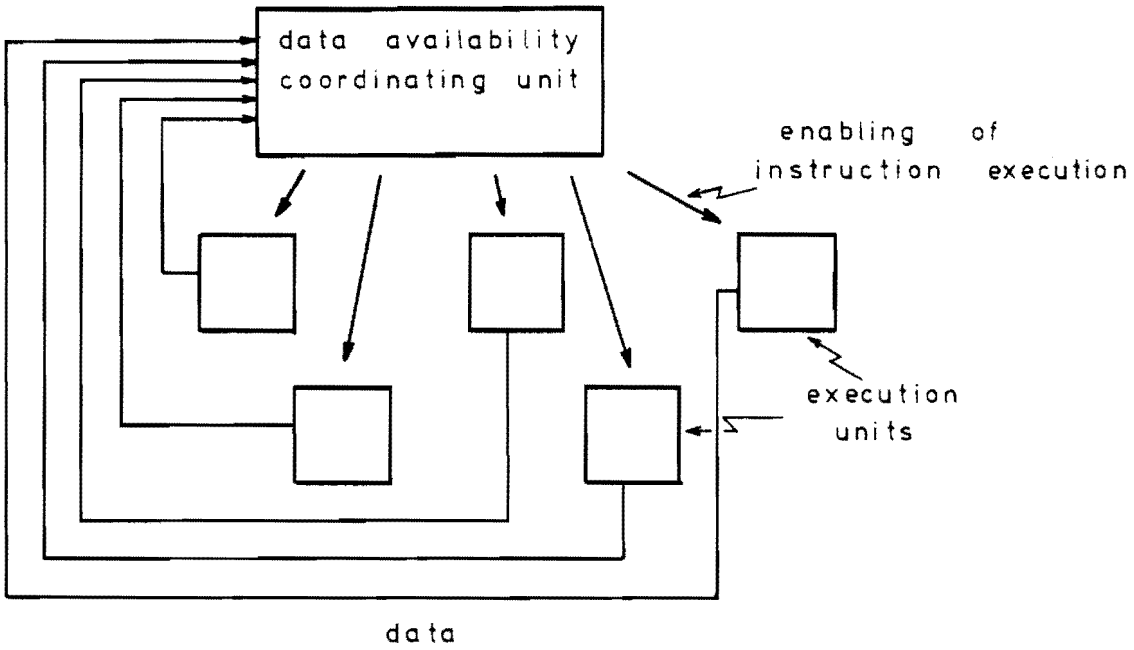


Figure 3.6: Conceptual Data Flow Computer

The approach is suited to exploitation of any level of parallelism in the same way as it is observed in associated data flow diagrams. The approach to management is very different from MIMD implementations in that problem oriented hardware assistance is provided.

Although a data flow processor would conceptually run very efficiently, constrained only by the dependence observable in data flow diagrams, its practical implementation is difficult. One problem noted (Agerwala and Arvind, 1982) is that of controlling and supporting large amounts of interprocessor communication. An example of a practical research oriented system is given by I. Watson and J. Gurd (1982). The cyclically operating structure developed is constrained in speed by the

components used, and the paper concludes that the potential of data flow hardware is not yet known.

3.3 Selection of Appropriate Form for Transient Stability Analysis

Being so widely accepted, the von Neumann model of computation has influenced the development of the raw materials for processing hardware. Typical items include sequentially addressed random access and read only memory, and central processing units embodying the approach. Consequently, in considering the development of a multiprocessor, the free availability of suitable components provides a strong impetus for selection of a von Neumann approach. Although data flow processors could utilise the available components at least some elements would require special development.

In implementing transient stability analysis programs on a multiprocessor an essential decision to be made is whether SIMD or MIMD architecture is most suitable. SIMD systems have advantages in price, simplicity, and availability whereas MIMD processors are better able to exploit parallelism. That is, for a given number of processing elements with similar speeds, it is likely that an MIMD system will achieve higher speed than an SIMD one, especially if a problem is ill-conditioned for SIMD processors.

3.3.1 Identification of Requirements

In Chapter 2, the two main program sections to be implemented in parallel were identified as:

.the solution of linear equations describing networks, and

.the solution of independent (unrelated) generator models.

Both of these areas have features which are poorly matched to the requirements for high throughput in SIMD implementations.

.The linear equations describing networks are represented using large, very sparse matrices. As a result, solutions involve operations on very sparse vectors which are difficult to distribute efficiently among processing elements.

.Generators are modelled by a variety of sets of sequentially executed code involving scalar operations. Parallelism in generator models is clearly observable when considering the independence of these sets. However, to utilise this parallelism requires an ability to execute parallel processes above the instruction level.

Qualitative consideration, as above, of the applicability of SIMD systems is an aid to choice between MIMD and SIMD implementations. Further useful evidence is provided by the practical implementation described in the following section.

3.3.2 Investigation of an SIMD Implementation

A number of investigations have been aimed at establishing the potential of SIMD processors in power systems analysis applications eg. references (Podmore et al, 1979), (Happ et al, 1979), (Orem and Tinney, 1979), and (Pottle, 1980). Of these, only the one by H.H.Happ et al provides sufficiently detailed results to compare actual performance with the maximum possible throughput. They used a CRAY-1 computer to simulate a 443-bus test power system network. Their objective was to illustrate the effective processing rate achieveable with little concern for how that related to the CRAY-1's capabilities. The concern here, on the other hand, is to determine the degree to which processing elements are used

effectively.

To improve efficiency various solution schemes were applied. All were attempts to redistribute elements within the matrices describing the system such that vectors which were operated on became less sparse. In two cases (ie. banded Matrix and banded BBDF) reordering was employed. These had no effect on numerical results. Reduction of the number of nodes for which states were evaluated was used in the other two cases (ie. Sparsity Reduction and Full Reduction). This can be seen as a simplification of the problem and, as such, could result in reduced execution times anyway ie. in serial execution. Note that similar schemes are considered in Chapter 4 as a means of improving MIMD operation. An unfortunate side effect of all of the approaches used is fill in of sparse matrices resulting in an increase in the total number of executable tasks. To account for the distortions introduced by fill in, execution time could be used to relate performance of the various approaches. However, a more valuable measure is the processing efficiency ie. the speed achieved, allowing for fill in overheads, related to the maximum throughput of the computer. For the CRAY-1 maximum throughput is 140 MFLOPS*.

The performance of a multiprocessor, with any number of processors, will exceed the performance of a single processor by a factor called the 'effective number of processors'. This can be compared with the number of real processors to determine useful processor operation. In a paper describing the CRAY-1, R.M. Russell (1978) states that the maximum number of simultaneously operating processors during vector operations is 64.

---

* - this figure is loosely based. Russell (1978) suggests a sustained throughput of 140 MFLOPS, with bursts of up to 250 MFLOPS, is possible.

This figure provides a useful basis for comparison with MIMD implementations as the effective number of processors can be established.

In table 3.1, both the normalised effective MFLOPS rate and the effective number of processors are given. As it uses no special redistribution of elements, the 'Tinney order 2' system provides a control example for comparison of the reordering and reduction schemes. Results are only presented for the execution of the substitution steps in solution.

Solution Approach	Real MFLOPS	Processing Efficiency	Effective Number of Processors
Tinney Order 2	3.4	2.4%	1.5
Banded Matrix	7.1	2.0%	1.3
Banded BBDF	17.3	4.6%	2.9
Sparsity Reduction	4.2	6.4%	4.1
Full Reduction	85.0	14.6%	9.3

Table 3.1: Execution Performance When Using an SIMD Processor

Using the Tinney order 2 approach, performance was very poor. Only 2.4% of available execution capabilities could be used effectively. Marginal improvements were achieved using redistribution schemes and the best improvement was gained using reduced matrix approaches. As mentioned, however, these could be considered an unfair comparison as some information is discarded eg. in the full reduction case only 90 of the 443 nodes remained. The high level of fill in occurring, especially in the full reduction case, is indicated by the great increase in MFLOPS for a less significant gain in performance.

Ignoring the reduction cases, the effective number of processors is

disappointing considering that there are 64 real processors available. Even an increase of a single order of magnitude appears to represent a daunting problem. With reduction it would appear, though, to be possible.

3.4 Conclusions

A variety of hardware approaches to multiprocessing have been outlined. Of these, SIMD and MIMD types were considered for suitability to transient stability analysis. SIMD systems were shown both qualitatively and quantitatively to be poorly matched to the needs of the problem.

MIMD systems offer a variety of possibilities for performance improvement over that achieved by SIMD processors. Qualitative consideration illustrates the scope for exploitation of parallelism in the execution of generator models. The scope in the solution of network models is less easily viewed. However, many recent publications describing MIMD approaches to the solution of sparse linear equations indicate significant performance improvement in comparison with the SIMD results presented in this chapter. It is left to the next chapter to consider these methods.

As mentioned in Chapter 1, the single objective throughout this thesis is achieving a means to high speed performance through high numbers of effectively operating processors. With this criteria MIMD approaches emerge as a clear choice. However, likely real implementations may require consideration of both the cost and simplicity of implementation so relative weightings for cost and speed would have to be established.

CHAPTER 4

PARALLEL SOLUTION OF LINEAR EQUATIONS

4.1 Introduction

The solution of linear equations forms a bottleneck in the parallel execution of transient stability analysis programs. Overall performance is consequently very sensitive to the computational efficiency achieved in execution of this section.

In 1980, Wing and Huang demonstrated that the level of parallelism existing within linear solution programs is very high, far higher than any previously reported implementations had indicated. However, they did not consider the degradation in performance likely if their ideas were practically applied.

This chapter outlines a variety of techniques, for the parallel solution of linear equations, which have been considered and, in some cases, applied. Following on from this, a description of a new approach, called the Parallel Bifactorisation (or PBIF) algorithm, is given. This solution method is based on the ideas of Wing and Huang, but is aimed more specifically at efficient practical implementation. This is achieved through a compromise between exploitation of parallelism and overheads due to operation organisation.

4.2 The Problem Specified

With a view to parallel implementation, there is little difference between the forward and backward substitution phases when using the LU factorisation method. The presence of diagonal elements throughout the

forward steps does not prevent almost identical definition of the phases. Therefore, to compact descriptions in this chapter, consideration is limited, where practical, to forward substitutions. Minor differences arising in backward substitutions are mentioned where appropriate.

Four representations of the forward substitutions steps are now presented. This seemingly repetitive approach is justified by the value each representation has in different aspects of the discussions which follow.

As described in Appendix 3, the forward substitution steps determine the solution, z, in the equation:

$$Lz = b \tag{4.1}$$

Because L is lower triangular, z can be found directly by the solution, in order, of the following equations:

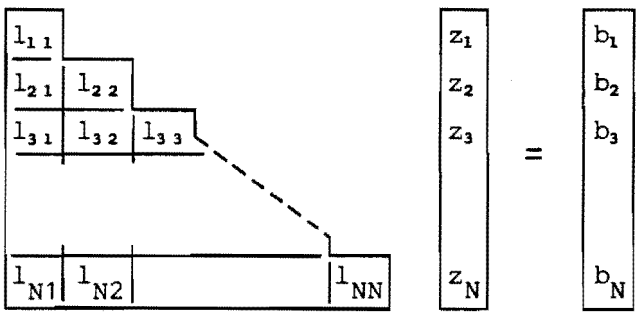
$$\begin{aligned} b_1 &= l_{11}\ z_1 \\ b_2 &= l_{21}\ z_1 + l_{22}\ z_2 \\ b_3 &= l_{31}\ z_1 + l_{32}\ z_2 + l_{33}\ z_3 \qquad (4.2) \\ &\vdots \\ b_N &= l_{N1}\ z_1 + l_{N2}\ z_2 + \quad . \ . \ . \ . \ . \end{aligned}$$

The solution steps can be more easily viewed in the following explicit restatement:

$$
\begin{aligned}
z_1 &= (b_1 &&)/l_{11} \\
z_2 &= (b_2 - l_{21}\ z_1 &&)/l_{22} \\
z_3 &= (b_3 - l_{31}\ z_1 - l_{32}\ z_2 &&)/l_{33} \qquad (4.3)
\end{aligned}
$$

$$
z_N = (b_N - l_{N1}\ z_1 - \quad \cdot \quad \cdot \quad \cdot \quad l_{N,N-1}\ z_{N-1})/l_{NN}
$$

Note that a high proportion of the elements of L are zero. The topological view of equation (4.1) is useful in observation of the distribution of non-zero elements.



(4.4)

Each element of the L-matrix can be associated with an update operation in the solution of equation (4.3). Off-diagonal elements are mapped to a multiplication and a subtraction while diagonal elements correspond to a division*. An example is given in equations (4.3) where off-diagonal element l_{31} is multiplied by z_1 and the result subtracted from a running total. This one to one correspondence between elements of L and executable tasks is useful in algorithm descriptions.

Finally, the substitution sequence can be represented in a data flow

---

* - As execution is faster, these divisions can be translated to multiplication by stored reciprocals.

format. In the diagram in figure 4.1 nodes, which correspond to executable tasks, are identified by indices corresponding to elements of L. A task graph depicting both the forward and backward substitutions is given in Appendix 6.

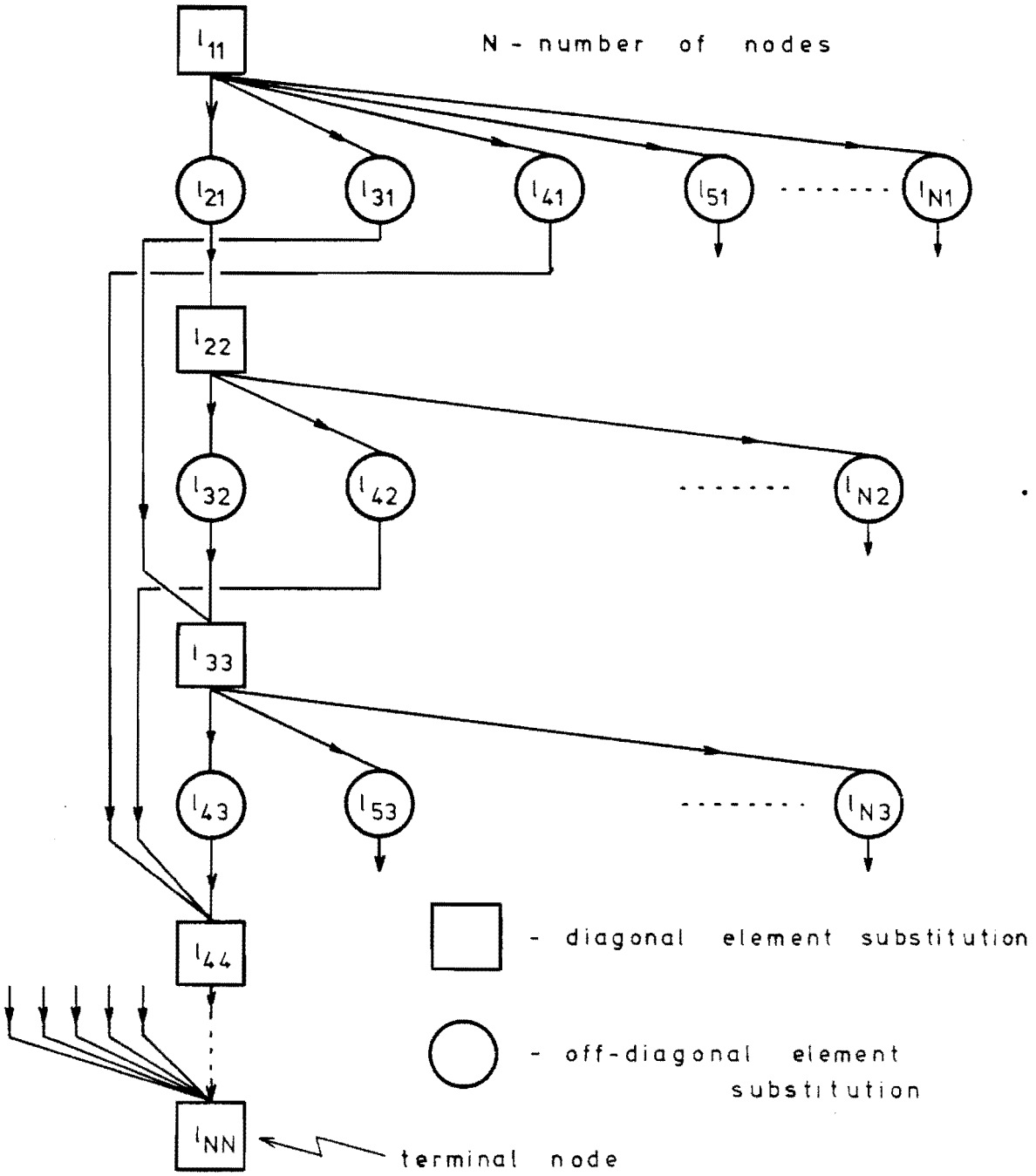


Figure 4.1: Forward Substitution Steps

4.3 Algorithmic Developments

A variety of approaches to MIMD implementation of the solution of sparse linear equations has been reported. Chronologically, the tendancy has been towards increasingly rigorous examination of the problem for scope to exploit parallelism. This search has had to be balanced against the practical need for reasonable consequent overhead levels.

Methods which improve throughput by reducing the quantity of information produced, even without loss of accuracy, are not considered here. However, nodal reduction, for instance, has been applied (Brasch et al, 1978) and shown to have interesting and useful properties when implemented on a multiprocessor.

4.3.1 The Sequential Method

This approach was developed during early EPRI sponsored work and is described by F.M. Brasch et al (1978). Its operation can be outlined in terms of the set of equations (4.3). The solution processes associated with these equations, each of which corresponds to a row in L, are distributed evenly among the processing elements. The processor associated with the first row solves for z_1 which it then broadcasts to the other processors. All processors then update their estimates of elements of z in parallel by executing substitutions involving z_1. Subsequently, z_2 can be determined, broadcasted, and substituted. This process continues until solution is reached. The performance expected from the sequential method is briefly examined in section 4.3.3.1.

4.3.2 Block Oriented Schemes

Many approaches to the parallel solution of linear equations are based on identification of parallelism from a topological view of the

original network description and L matrices. Examples are described in papers and reports by: Hatcher et al (1977), Conrad and Wallach (1977), Pottle and Fong (1978), Fong (1978), Brasch et al (1978 and 1979), Kees and Jess (1980), and Fawcett and Bickart (1980).

The principle of operation of these techniques is the creation of clusters of elements of L which are independently solvable, and can thus be distributed among processors. Figure 4.2 illustrates a way in which this can be achieved. A matrix in Block Bordered Diagonal Form (BBDF) is shown with four clusters. The physical network clustering is related to the matrix topology. Solutions related to each diagonal block (labelled 'F') can be carried out simultaneously. In addition, once substitutions on each diagonal block are completed, updates due to elements in blocks in the lower rows (marked 'B') can go ahead. Hence, all operations not associated with elements in the cut-set block (labelled 'C') can be carried out in parallel.

Problems reducing the efficiency of block approaches include:

- fill in due to the reordering required to create clusters, and
- the serial solution related to elements in the cut-set block. This unavoidable block represents connections between the clusters. Generally, increases in the number of clusters (to allocate processing among more processors) will also result in an increase in the size of the cut-set block.

Variations from the BBDF, aimed at improved performance, have been described. For example, the Nested Block Diagonal Bordered Form (NBDBF) described by Pottle and Fong (1978) creates further sub-blocks within each block. This 'recursive clustering' could be applied on hardware with bus

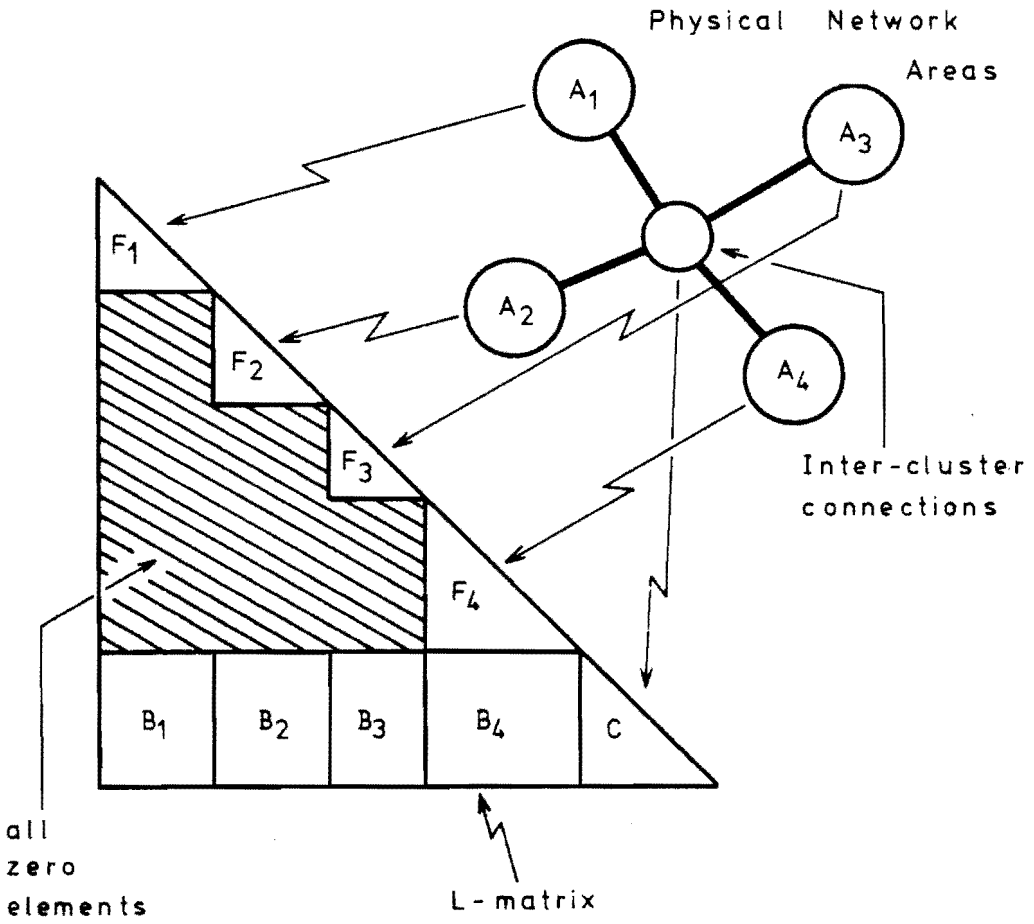


Figure 4.2: Clustering for Block Schemes

switches (see section 5.3.1) to enable efficient use of communication resources. Individual blocks could be solved with the switches open and the cut-set solved with them closed.

4.3.3 Elemental Approaches

Rather than heuristic creation of parallelism through topological clustering, parallelism can be identified better and used to more advantage using data flow descriptions. Wing and Huang (1980) presented a detailed analysis of the parallelism available throughout the processes involved in triangulated linear equation substitutions. Their approach is centred on task graphical methods. The task scheduling technique proposed is based on Hu's algorithm (Hu, 1961). The strategy used, called the modified Hu level

scheduling scheme, is as follows. (Note that the approach can be viewed in the context of a data flow description (figure 4.1)):

.among the ready tasks, select the one with the smallest level number* for assignment to any available processor first.

.if two of more tasks are tied, select the one with the most successors first.

They show that, although not guaranteed optimal, this approach is very efficient in utilisation of parallelism. In their evaluation of the effectiveness of the algorithm unit execution time was assumed and management overheads were ignored. Their results, based on matrices which were not specially reordered, showed a significant improvement over block schemes. The performance estimates presented (see Chapter 9) provide an upper bound towards which any practical schemes might be aimed.

Two practical elemental schemes are reported here. The first was described by Brasch et al (1981) in an EPRI report and is briefly outlined here, while the second, called the PBIF algorithm, is described by Arnold et al (1982) and is presented here in greater detail. Both are based on similar raw material but the implementations differ significantly. Selection of the more efficient scheme is hampered by many factors including differences in the hardware assumed in their respective performance evaluations.

---

* level number is defined as 'the latest time by which' the node concerned 'must be processed in order to complete the task graph in the minimum time'

4.3.3.1 EPRI Methods

The elemental schemes described by Brasch et al (1981 and 1982) will be referred to as the EPRI algorithms. Three approaches, differing in the minimum size of task, were examined: non-switching, serial switching, and parallel switching. These vary in both degree of utilisation of parallelism and need for inter-processor communication. Strictly, Hu's algorithm can only be applied exactly to the parallel switching method which also has the highest communication requirement. A variety of scheduling schemes, originating from Hu's ideas but aimed at efficient practical operation, were considered for all of the approaches. It was concluded that, because of its lower communication requirements, the non-switching scheme was most promising.

The groups of updates forming tasks in the non-switching algorithm correspond to rows in the L matrix ie. processors are assigned all of the substitutions in row 'i', for instance, culminating in the determination of z_i. In figure 4.3, a typical task is indicated within a task graph which is rearranged from figure 4.1.

It is possible that, at an instant, some but not all updates within a task are ready as each update has a different predecessor. Therefore, execution of a task can begin and be held up part way through. This, among other problems, results in complicated scheduling schemes. A further difficulty is the practical determination of task readiness which requires up to date information with respect to the status of all task predecessors.

Fixed task execution times are not assumed so tasks can run assynchronously. This is an important practical feature as it avoids idling while waiting for the slowest processor at each step. However, it also means that distribution of tasks among processors cannot be determined

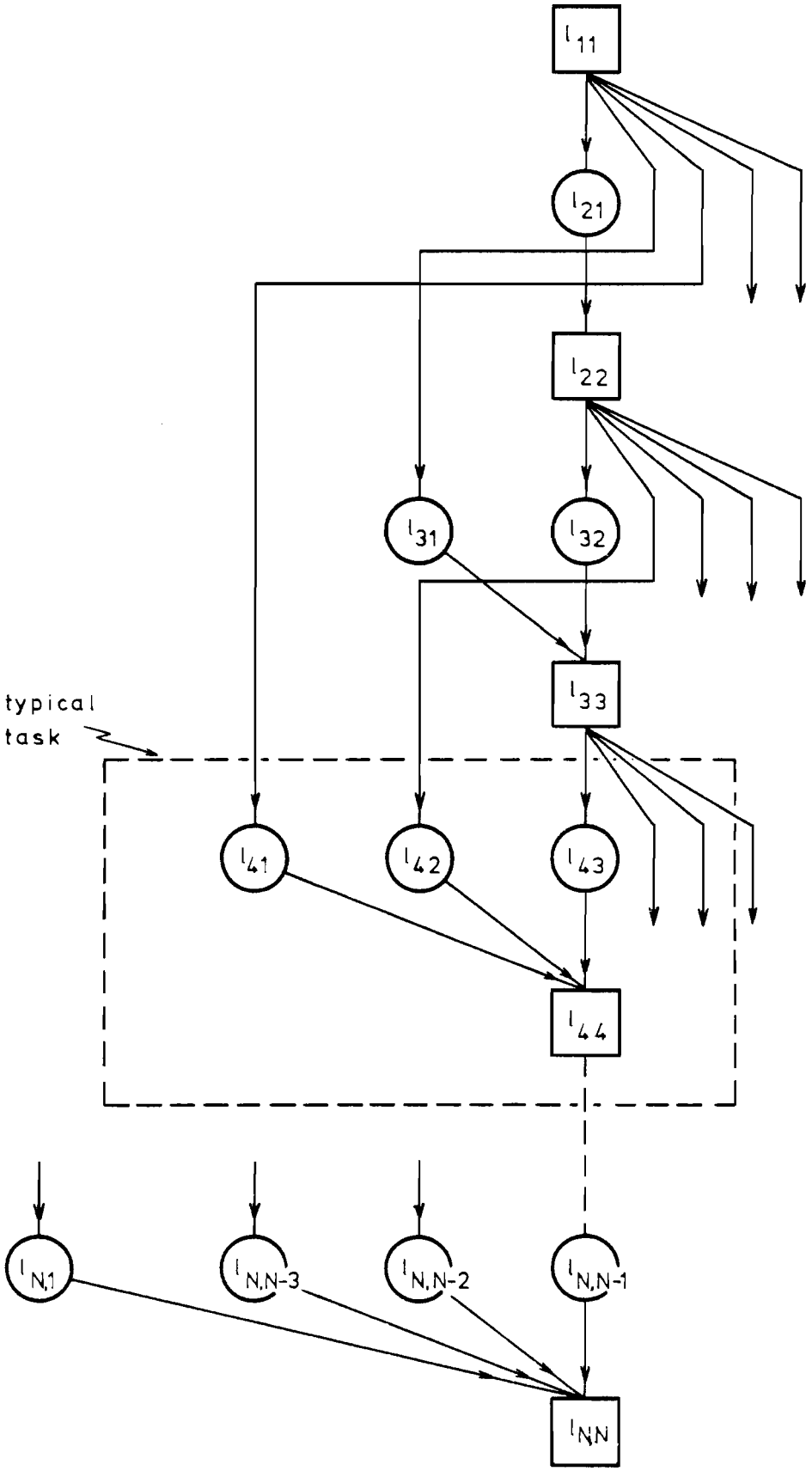


Figure 4.3: Task Arrangement for Non-Switching Scheme

before run time and, consequently, that scheduling must be dynamic.

Without considering communication needs, the gains in performance of the EPRI scheme over non-elemental approaches are illustrated in figure 4.4 which was taken from the EPRI report. Improvements by factors of 4 and 10 respectively over the saturated block and sequential performances are mentioned. However, performance estimates generated by a more detailed simulation suggested a considerable drop in performance. For example, with 50 processors the non-switching scheme dropped from an estimated 60.7% to 29.7% effective processor utilisation. Performance, however, is still better than that for non-elemental methods.

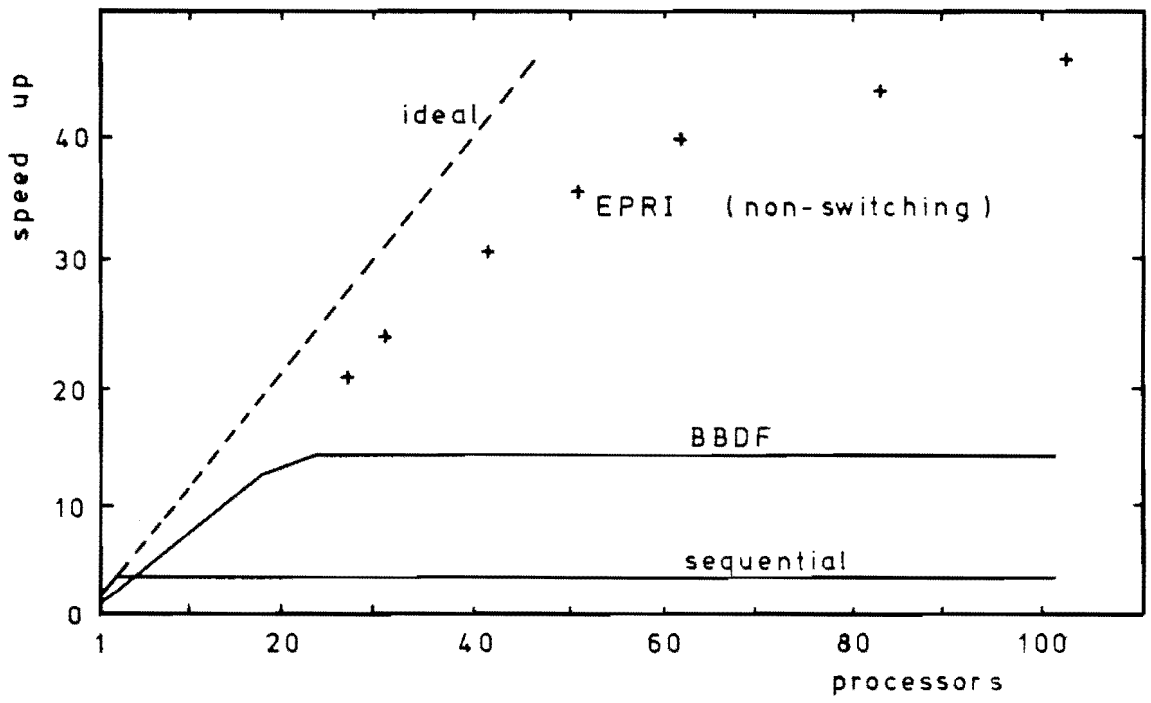


Figure 4.4: Comparison of Elemental, Block, and Sequential Approaches

4.4 The PBIF Algorithm

The solution approach presented here is based on identification of processes which will lead to efficient execution. The natural structure of the problem is exploited in the selection of appropriate tasks. The

resulting solution method, called the PBIF algorithm, utilises a high proportion of available parallelism without unreasonable management overheads.

Hu's method, which is applicable to a wide range of task graph forms, and doesn't take account of management and task distribution, is used only as a guide and is not implemented. Instead, scheduling is arranged using a very quickly executed approach matched to the form of the problem.

4.4.1 Task Identification

Nodes, within a task graph, have the following properties:

In(i,j): the number of in-edges of node(i,j)

Pr(i,j): the set of all predecessors of node(i,j)

DYNIN(i,j): the number of elements of Pr(i,j) whose associated tasks do not have the completed state. Note that this is a dynamic property.

A task graph depicting the substitution of the elements in the L-matrix was given in figure 4.1. The following properties of LU substitions are observed from the task graph.

$$Pr(i,j) = node(j,j);\ In(i,j) = 1 \qquad (4.5)$$

$(i \neq j$, column j, all i that exist)

$$Pr(i,i) = nodes(i,j) \qquad (4.6)$$

$(i \neq j$, row i, all j that exist)

The property given by (4.5) offers a considerable saving in the management required to determine 'ready' status of all off-diagonal tasks. That is, once a diagonal task has 'completed' status all the off-diagonal

tasks in that column must be ready.

Hence, an opportunity is presented to determine the status of groups of updates through a single check. Advantage is taken of this possibility in the PBIF algorithm. For this new algorithm, the basic process consists of a serial sequence of updates involving the diagonal element followed by the off-diagonal elements in a column of L. The same is true of the backward substitution steps except that there is no processing associated with diagonal elements. This column oriented update grouping in task formation is similar to that proposed by Wallach and Conrad (1981). They use column based blocks which are distributed one to each processor. Columns are solved serially, but blocks within columns are solved simultaneously.

Process management involves detecting which diagonal nodes have 'ready' status. This state is indicated, using property (4.6), when all nodes in the row are 'completed' which corresponds to the instant when, for a diagonal node (j,j):

$$DYNIN(j,j) = 0 \qquad (4.7)$$

To implement this management function, DYNIN is stored as a vector with each element corresponding to a diagonal element in L or U. The initial value of DYNIN is set by :

$$DYNIN(j,j)initial = In(j,j) \qquad \text{(for all j)} \qquad (4.8)$$

During execution the contents of DYNIN are updated after each node is substituted. For node (i,j)

$$DYNIN(i,i) = DYNIN(i,i) - 1 \qquad (4.9)$$

In a practical implementation, the functions of updating and observing the state of the DYNIN vector ((4.7) and (4.9)) could be handled in various ways. The method investigated involves each processor supervising itself by searching DYNIN for ready tasks. One alternative is to assign a specific processor to searching DYNIN.

4.4.2 Scheduling

In the previous section the groups of substitutions forming tasks were defined. At points during execution, the order in which these tasks are allocated to processors, and to which processor, must be determined. For high effective processor utilisation the scheduling scheme should be balanced between simplicity and hinderance to exploitation of parallelism. The scheme implemented is simple and it schedules tasks using priorities which result in organised and effective distribution.

Tasks are arranged in order of ascending column number ie. those furthest to the left in the L matrix (or to the right in the U matrix) are given the highest priority. This is similar in principle to Hu's algorithm as can be seen in the task graph (figure 4.1): The tasks furthest from the terminal node are scheduled with the highest priority.

In addition to the need to determine the best ready task to assign to a processor, a desirable scheduling feature is minimisation of the search length ie. the time taken to find any ready tasks. In the PBIF algorithm this is achieved by initiating the search at places where ready tasks are most likely to exist. For the column based substitution tasks used, the most likely ready tasks correspond to those columns just after (ie. to the right of, for the L matrix) the ones already assigned.

The execution time devoted to searching is an overhead if one or more tasks is ready. In addition, the use of common resources implied during the searches can impede the useful operation of other processors. Hence, performance improvement through appropriate choice of two parameters within search routines is considered. The first is the time between checks for the readiness of each task. Increasing this time reduces demand for common resources, ie. memory access, but it also increases the time taken to find available tasks. The second parameter is a restriction to the number of tasks considered for readiness. In this way, searching can be concentrated on those tasks most likely to become ready, but some possibly ready tasks are ignored. In Chapter 9 practical attempts to tune these parameters are described.

4.4.3 Process Definition

For the implementation investigated all processors are assumed to be set up with identical local program code. Tabular descriptions of the dynamic state of tasks, which are needed for scheduling, are accessed and updated by every processor. Until execution is complete all processors assynchronously search for ready tasks which, if found, are executed. No single processor has a special purpose so the sharing of load is very even.

A process consists of the three sections shown in figure 4.5. The symbols used are defined in Appendix 3.

.Search

The objective here is to find a 'ready' diagonal node. This is achieved by observation of the DYNIN vector awaiting condition (4.7). Having located an element satisfying this condition the column associated with that element is selected for substitution. To inform other processors that the column has been selected a marker

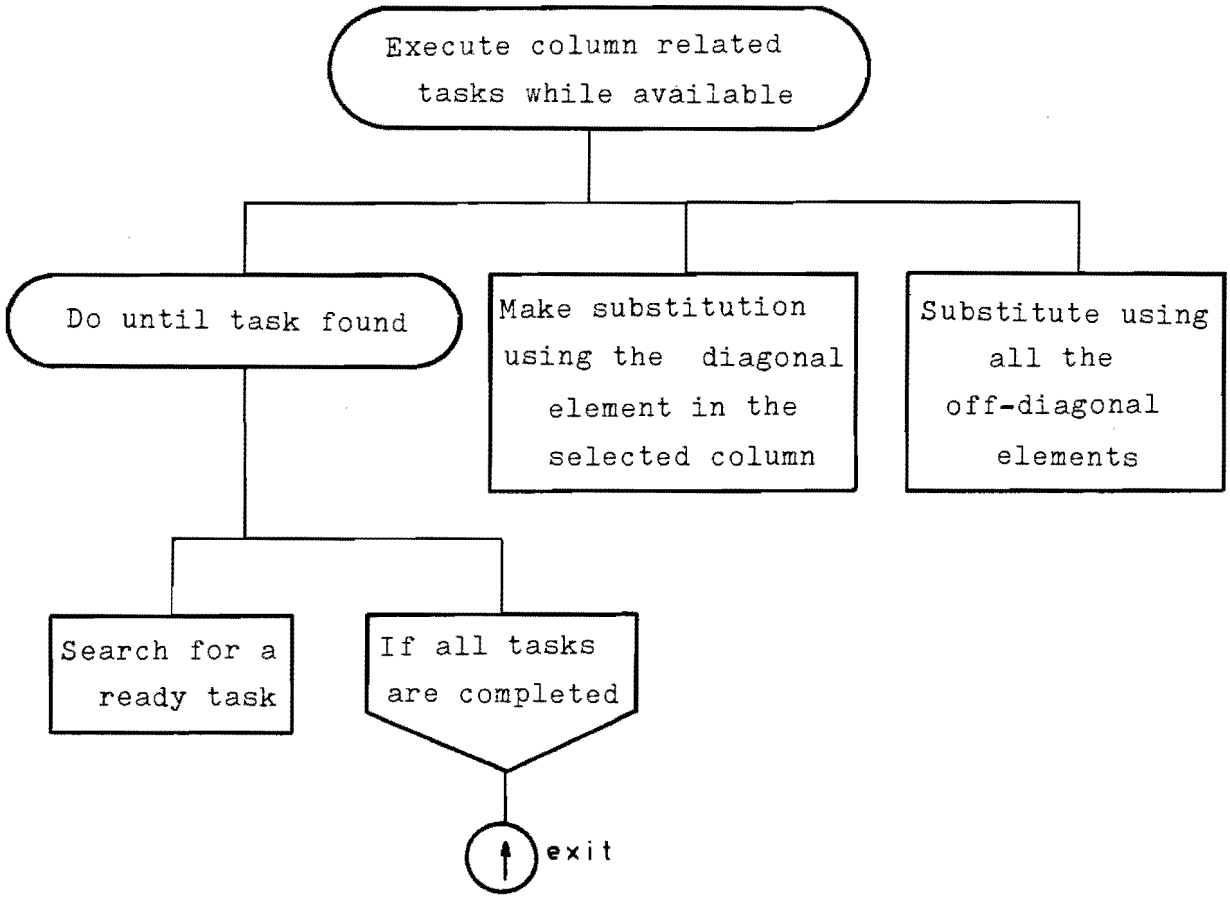


Figure 4.5: Execution Sequence for an Individual Processor

value is substituted in DYNIN indicating that the state has changed to 'running'.

A diagram depicting the execution sequence of a suitable search routine is given in figure 4.6.

.Diagonal Update

For any element 'i' in z the diagonal update involves:-

for L : z(i) = z(i)/l(i,i)

for U : x(i) = z(i)/1

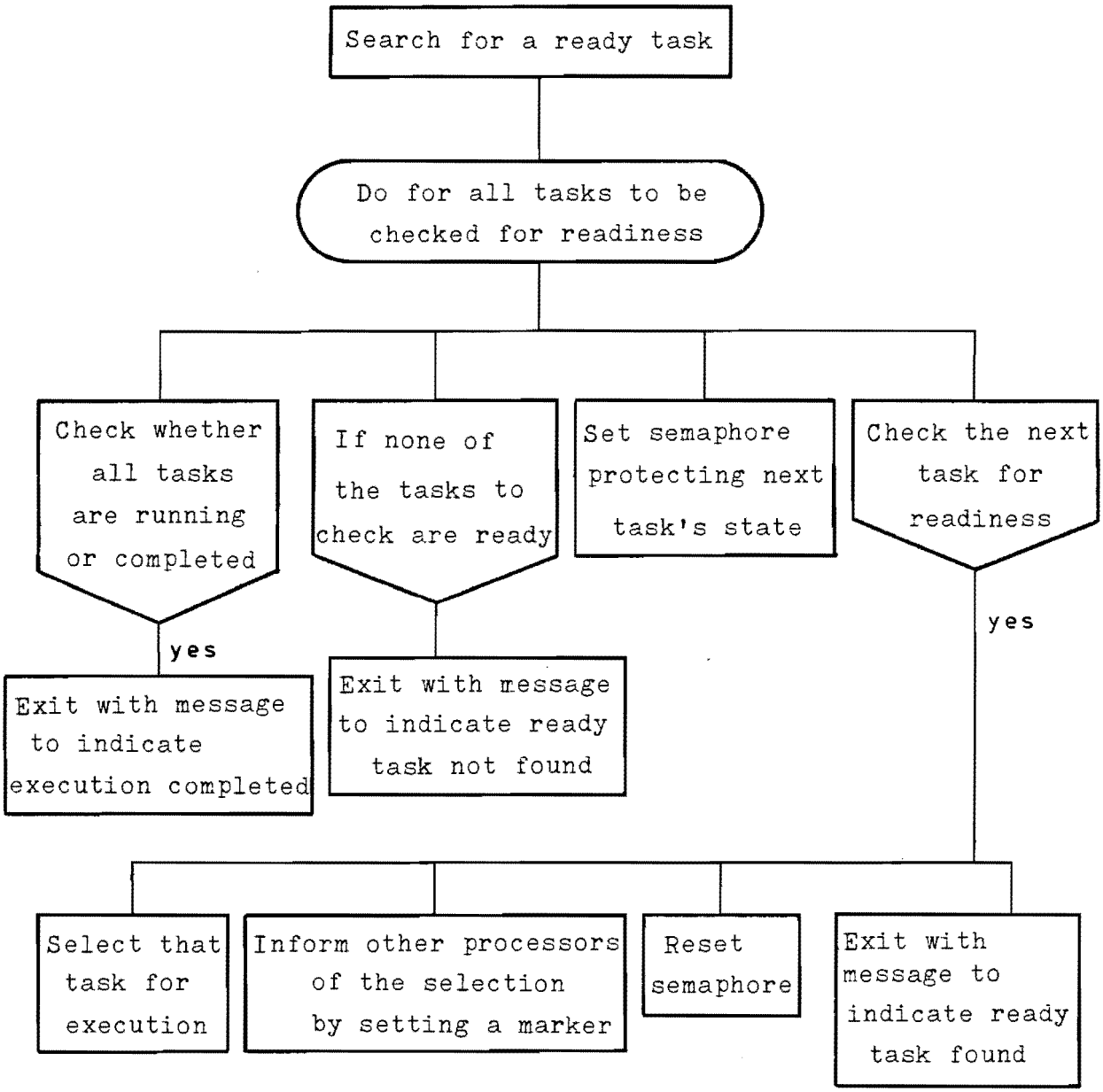


Figure 4.6: Execution Sequence to Find a Ready Task

.Column Substitutions

for L : for all non-zero elements in column j

$$z(i) = z(i) - z(j)*l(i,j)$$

DYNIN(i,i) = DYNIN(i,i) - 1

for U : for all non-zero elements in column j

z(i) = z(i) - x(j)*u(i,j)

DYNIN(i,i) = DYNIN(i,i) - 1

Off-diagonal substitutions are made in order of proximity to the diagonal. The contents of DYNIN must be reinitialised before both the forward and backward substitution steps.

4.4.3.1 Use of Semaphores

Each element of z and DYNIN is a resource common to all processors and can be updated by any processor. To maintain security during updates a semaphore must be associated with each element of z and DYNIN. The period during which semaphores are set should be minimised to reduce delays imposed on the operation of other processors. An application of this principle is illustrated in figure 4.7. In updating the elements of B, increments are calculated before the semaphore is set and protection is only provided during the short period while these increments are added to the running total.

For practical implementation, the use of semaphores requires a bus locking capability. That is, a single processor must be able to stop any other accesses to the location containing the semaphore while it checks and/or sets the value.

4.4.3.2 Coding

Listings of sections of the program executed by each processor are given in figure 4.7. Written in PL/M-86, they implement both the search and substitution phases during the forward factor matrix substitutions of

```
/* The following routine searches a list of tasks which are likely to
        be ready during forward substitution steps in application of the
        bifactorisation method. It exits with status describing the
        success of the search :
                ENDED - all substitution tasks have been allocated
                STILL GOING - a task has been successfully found
                TASK NOT FOUND - no task found, but could try again */

SEARCH: PROCEDURE BYTE;          /* typed procedure returning status */

DECLARE NUMBER INTEGER;          /* local temporary variables */
DECLARE I INTEGER;

        I = 0;          /* counter for tasks examined */

        DO FOREVER;
                /* Check whether all tasks allocated */
                IF ((NEXT$COLUMN > LENGTH$B) OR (FORWARD$DONE = TRUE))
                                        THEN RETURN ENDED;
                /* Check whether all tasks searched this time */
                IF I > SEARCH$LENGTH THEN RETURN TASK$NOT$FOUND;
                IF ( I + NEXT$COLUMN ) > LENGTH$B THEN
                                        RETURN TASK$NOT$FOUND;
                /* Select a new task to check */
                J = I + NEXT$COLUMN;
                /* and set a semaphore to say busy */
                PSEM B$SEM(POSN$IN$B$BEFORE$ORDERING(J)) SET_;

                /* See if the task is ready */
                IF (NUMBER := NUMBER$IN$ROW$FOR(J)) = 0 THEN DO;
                        /* leave a message for others */
                        NUMBER$IN$ROW$FOR(J) = DUMMY;
                        /* remember which one it is */
                        COLUMN$NUMBER = J;
                        /* release the semaphore and exit with success */
                        VSEM B$SEM(POSN$IN$B$BEFORE$ORDERING(J)) RESET_;
                        RETURN STILL$GOING;
                        END;

                /* If this is the first task see if it has
                                                been allocated */
                IF ((NUMBER <= DUMMY) AND (I = 0)) THEN DO;
                        /* If so, update the start of the list */
                        /* NEXT COLUMN is the column up to which all
                                tasks have definitely been allocated */
                        PSEM NEXT$COLUMN$SEM SET_;
                        NEXT$COLUMN = NEXT$COLUMN + 1;
                        VSEM NEXT$COLUMN$SEM RESET_;
                        END;
                /* otherwise just go on with the next task */
                ELSE DO;
                        I = I + 1;
                        END;

                /* release the semaphore */
                VSEM B$SEM(POSN$IN$B$BEFORE$ORDERING(J)) RESET_;

                END;
        END SEARCH;
```

continued over page

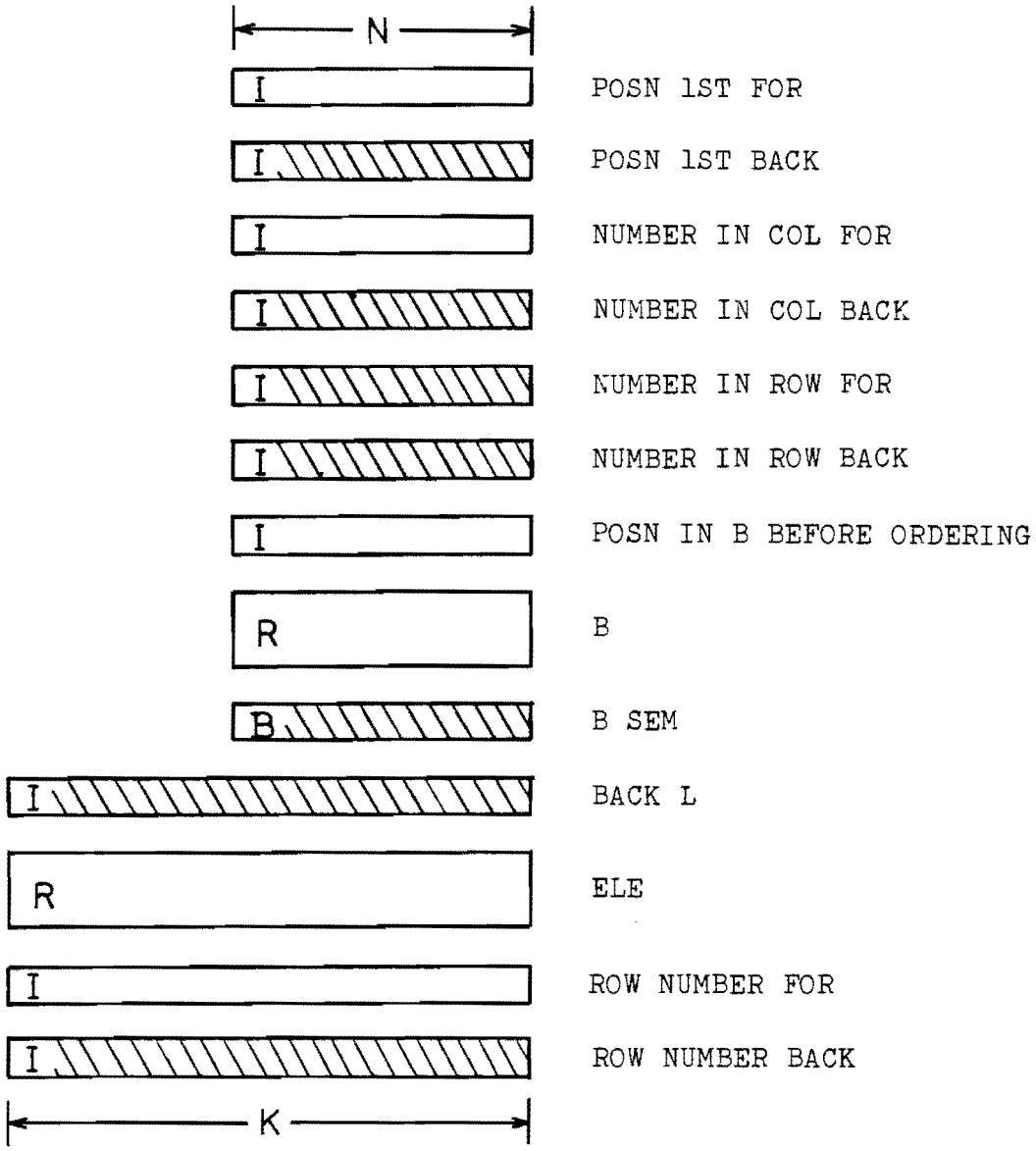
```
/* Forward substitution steps */

FORSUB: PROCEDURE;

DECLARE I INTEGER;      /* local variable */

        /* Take the element of B selected in routine SEARCH */
        Z.RE = B(TEMP$INT := POSN$IN$B$BEFORE$ORDERING(J)).RE;
        Z.IM = B(TEMP$INT).IM;

        /* prepare for the diagonal update step */
        B(TEMP$INT).RE,B(TEMP$INT).IM = 0.;

        /* Make updates associated with all elements in the selected
                column */
        DO I = POSN$1ST$FOR(J) TO
                (POSN$1ST$FOR(J)+NUMBER$IN$COL$FOR(J)-1);

                /* Calculate the increments to be made */
                B$INCR.RE = Z.RE*ELE(I).RE - Z.IM*ELE(I).IM;
                B$INCR.IM = Z.IM*ELE(I).RE + Z.RE*ELE(I).IM;

                /* Apply the increments using semaphore protection
                        only where necessary */
                PSEM B$SEM(ROW := ROW$NUMBER$FOR(I)) SET_;
                B(ROW).RE = B(ROW).RE + B$INCR.RE;
                B(ROW).IM = B(ROW).IM + B$INCR.IM;
                NUMBER$IN$ROW$FOR(ROW) = NUMBER$IN$ROW$FOR(ROW) - 1;
                VSEM B$SEM(ROW) RESET_;

                END;
        END FORSUB;
```

Figure 4.7: Coded Implementation of the Search and Forward Substitution Routines

the bifactorisation method. Complex matrix elements are used and the symbols employed are defined in Appendix 4.

4.4.3.3 Memory Requirements

The extra storage required for information specifically related to multiprocessing is a cost to be taken into account in the design of parallel processing systems. Figure 4.8 illustrates the vectors used, as defined in Appendix 4. Comparison is made with those vectors needed on a single processor system to give some idea of the additional memory required. For example, using 32 bit floating point real number



I - integer
R - real
B - bit

N - number of nodes
K - number of non-zero elements in ELE

Note that vectors specifically introduced to suit a parallel processing environment are shaded.

Figure 4.8: Vectors and Storage Requirements for the PBIF Algorithm

representation, a 1000 bus system with a sparsity coefficient of 0.95 would require 373,000 bytes for data storage using this algorithm as opposed to 264,000 for a serial implementation. Power system problems of this size may well have sparsity coefficients in the order of 0.99 where the figures are 93,000 and 64,000 bytes respectively. Therefore, an increase in memory needs of about 50%, in comparison with serial implementation, is predicted.

4.5 Conclusion

A strong trend from block towards elemental schemes for the parallel solution of linear equations has arisen over the past five years. The trend has been well illustrated in the work by F.M. Brasch et al. In a 1977 paper (Hatcher et al) it was stated that 'parallelism exists below the block level, but the cost of implementing this low level of parallelism is higher than the gains'. By 1981 (Brasch et al), this opinion had been modified to say, with respect to block and sequential schemes, that 'while the results were not entirely negative, it was felt that the theoretical best case performance expectations were not really adequate'.

The tendency towards more rigorous searches for parallelism has led to great improvements in effective processor utilisation. However, practical implementation problems remain and selection of the best algorithmic approaches are still heuristically based.

Theoretical investigations of the performance achievable using elemental schemes are useful in that they have resulted in determination of the limits to performance that can be expected from any practical implementation.

A solution method, called the PBIF algorithm has been presented. In subsequent chapters its practical performance is compared with both other algorithms and the theoretical performance limits.

CHAPTER 5

EXISTING MIMD MULTIPROCESSING SYSTEMS

5.1 Introduction

Only a few highly parallel MIMD processing systems have been built or proposed and, of these, most have emerged over the past five to ten years. The term 'highly parallel' is used here to describe systems with a sufficient number of processors to result in at least an order of magnitude increase in speed when compared to the performance of the individual processors from which they are constructed.

Two factors have led to the low level of development of MIMD hardware when compared, for example, with the current proliferation of SIMD devices. Firstly, as discussed in chapter 3, the development of efficient software is considerably more complex, and secondly, hardware realization is much more difficult.

The complexity of processor memory interconnections increases rapidly with the number of processors employed. It has been suggested (Fairbourn, 1982) that the use of standard components in the building of highly parallel structures results in overwhelming difficulties in construction and maintenance, and, in addition, places restrictions on the architecture and capabilities of the system. It is stated that the use of specially designed VLSI components is the only available option in developing really practical systems. However, this technology is in its infancy and only pioneering research is in progress. Computer manufacturers are producing complex integrated circuits aimed at easing the problems of interconnecting processors and memory. For instance, INTEL

have recently introduced the iAPX-432 VLSI Micromainframe system. This includes a VLSI processor, the iAPX-43203 (INTEL(a), 1982), which allows the construction of bus switching arrays.

Hence, although the existence of VLSI-based highly parallel structures is foreseen, they are not yet available. Algorithms, suited to parallel processing are being developed in many areas requiring high speed computation in the expectation that suitable hardware will emerge. Because serial simulation of the execution of these algorithms is difficult to validate, testing and development using physical MIMD structures is desirable. Consequently a high proportion of existing systems were developed in research institutions primarily as tools to aid future development and have limited raw processing power. Very few commercial MIMD systems are available.

This chapter describes a range of existing MIMD systems. Features varying between systems include raw processing power and software development facilities. The essential characteristic of each system as described, however, is the bus structure. This information, and an outline of the practical needs of a system, which is given in Chapter 6, are used in the development of a new, research oriented multiprocessor, the UCMP system, which itself utilises a very flexible and efficient bus structure.

5.2 Bus Structures

Whenever two or more entities require the use of a single resource at the same instant conflict occurs and delays arise. In multiprocessing systems simultaneous requests for the use of either a data transfer path or a memory unit results in conflict and consequent performance degradation.

Serial processing systems use buses to interconnect a processor, memory, and I/O devices. Because a single element initiates transfers no

need arises for decision-making units to allocate use of the bus*. In addition, because bus cycles occur in well-defined time order, it is generally easy to determine the likelyhood of bottlenecks due to heavy bus utilisation, enabling the selection of an appropriate bus structure.

Like serial machines, multiprocessors use buses to interface processors, memory, and I/O devices, but the number of possible connections adds a new dimension to design problems. Factors influencing the choice of structure are wide ranging and a very important consideration is the program code to be executed.

5.2.1 Variety of Structures

The data transferred between processors and shared memory generally increases with the number of processors. Consequently, larger systems tend to require more complex bus structures capable of distributing the heavier traffic.

Interconnection methods vary between extremes. The simplest is the single time-shared bus, or ring network (Haynes et al, 1982). The most complex is the full crossbar network where a separate path exists between each processor and each memory unit. For full crossbar networks conflict can only occur in access to a single memory unit while for all other structures contention can also occur in obtaining a path to reach that memory.

The form of structures is conveniently divided into two classes. In

---

* - an exception to the synchronous nature of serial bus usage occurs when direct memory access (DMA) devices, which can use a bus without intervention from the processor, are included in a system.

one, all processors see uniform length paths, ie. experience similar delays, when accessing global memory. Such systems are classed as homogeneous. Alternatively, heterogeneous systems, such as tree structures, have path lengths dependent on the processor and memory involved.

5.2.2 Criteria for Comparison of Structures

Full crossbar networks require hardware in quantities proportional to the product of the number of processors and the number of memory units. (Frequently the same number of processors as memory units are used.) Such structures are impractical because of the number of interconnections required for realization of highly parallel systems. Hence, much research has been directed at the definition of the least complex bus structures possible while constrained by a variety of performance needs. Three parameters are used (Haynes et al, 1982) to categorise interconnection possibilities:

.Hardware complexity and cost including physical realizability and fault tolerance.

.the level of processor utilisation to be achieved ie. relating performance of systems to that of its constituent processors.

.the degree of specialisation ie. the level to which the final program to be executed can be defined.

Figure 5.1 (Haynes et al, 1982) is an attempt to illustrate in two dimensions the relative performance of various proposed networks, defined in the article, in terms of the above areas. The form of the individual networks is not important but the trend which emerges is of interest. The

implication, portrayed in the diagram, is that cost, generality, and efficiency must be traded off against one another. Of these it is perhaps most difficult to appreciate the effect of generality, the inverse of specialisation.

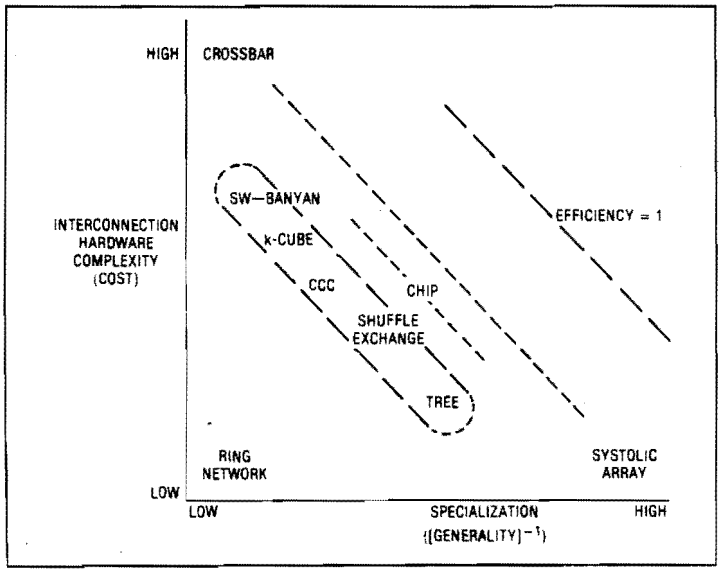


Figure 5.1: Processor-Memory Interconnection Design Options

5.2.3 Specialisation and Polymorphism

Since their conception, an inherent feature of computers has been polymorphism (Snyder, 1982). That is, their function can be radically altered by small changes such as the loading of a different program. The desirability of polymorphism led to the development of microprocessors in the early 1970's as reconfigureable logic circuits.

As suggested in figure 5.1 it is necessary to predefine the function of a multiprocessor in order to obtain efficiency in the most cost-effective manner. Therefore, polymorphism is expensive to achieve in multiprocessors. The reason can be seen by considering implementation of some new function. With a good knowledge of those paths between processors and memory which will be heavily utilised and those which will only

transfer light traffic, a detailed idea of a bus structure can be developed with little redundancy and resources placed where they are needed most. In practice, detailed knowledge of problems often leads to the grouping of processors to handle blocks of code with little outside interference. An example of such heterogeneous structures is the development of systolic arrays (Kung, 1982). In these, very efficient use is made of bus bandwidth by assigning data to tightly coupled groups of processors. Access to a central data storage facility occurs only at the extremities of groups.

A strong relationship exists between generality of application and homogeneity. The trend, evident in figure 5.1, is for homogeneity to increase with generality. For instance, systolic systems are the most specialised, followed by tree structures etc.. The full crossbar and ring structures are the least specialised and are homogeneous.

Many structures have been formed utilising various combinitions of the extremes which have been outlined. Selection of optimal structures remains is a heuristic task and work must still be done to formalise possible approaches.

5.3 Structures Adopted in Some Existing and Proposed Systems

Leading to the development of the UCMP multiprocessor this section outlines the structures of a number of existing and proposed systems. Some of the systems have been applied to power system related problems.

The figures accompanying the descriptions use the following symbols :-

P - to identify processors

M - to identify memory units

BS - to identify bus switches

Subscripts index processors and memory units.

5.3.1 Structural Proposals by Fong

J. Fong (1978) descibes two MIMD structures and relates them to use in the area of power system simulation. The first of these, structure A, illustrated in figure 5.2a has a simple structure utilising a single bus connecting a number of processing elements. This is extended, using switches in the interconnecting bus, to structure B, where it is possible to isolate groups of processors. This structure is shown in figure 5.2b.

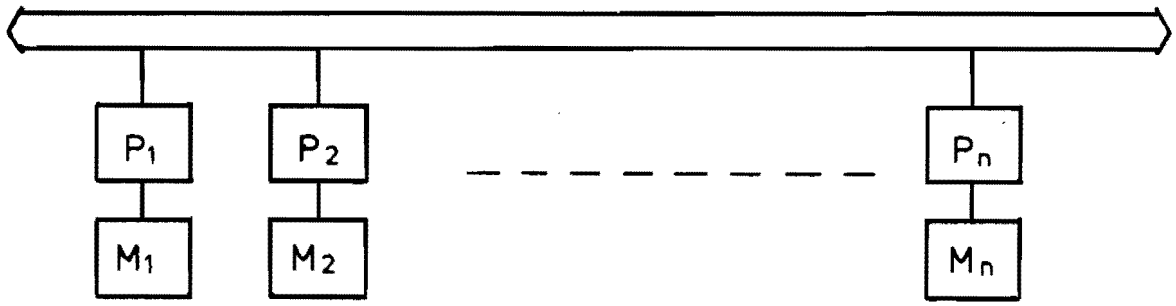


Figure 5.2a: Structure A (Fong)

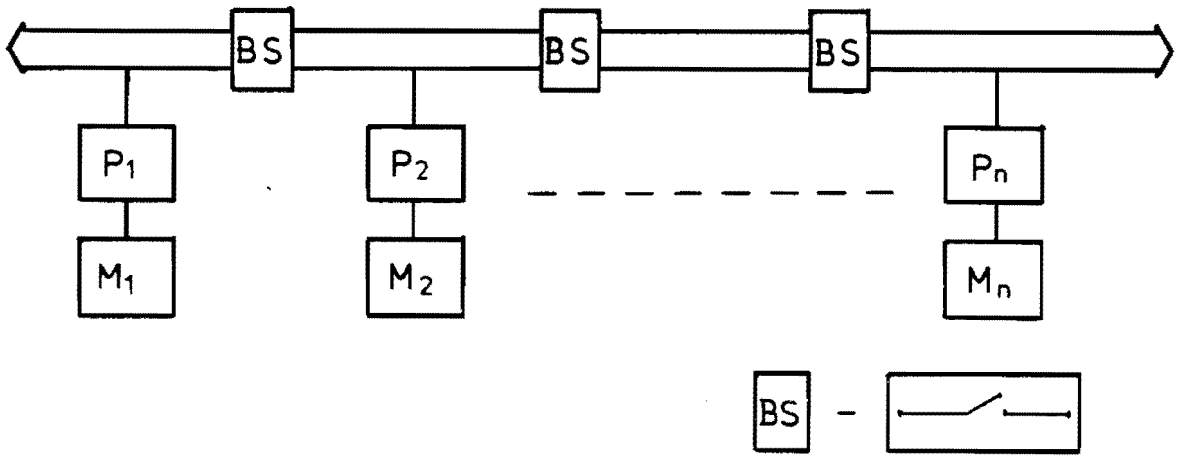


Figure 5.2b: Structure B (Fong)

In both structures each processor is accompanied by a memory unit which can contain both program and local data. The bus used allows transfers between processors rather than between processors and memory. Hence, when data must be distributed, both the sending and receiving processors must cooperate in the transfer. By operating more than one processor as a receiver simultaneously data broadcasting would be possible.

High bus conflict possibilities in structure A lead to the less vulnerable form used in structure B where more efficient use can be made of the bus. This is achieved by dynamic partitioning of processors into groups according to the needs of the problem. Two such groups can simultaneously use the bus if a bus switch between them is open. Bus switching operations could be controlled in many ways eg. by a single processor, or by processors in close proximity to individual switches.

Structure A is homogeneous. Structure B is slightly heterogeneous in that some processors can form groups with a topology which others could not form. The structures are not limited to the execution of specific problems and are of low complexity. Hence, with significant levels of bus traffic, poor efficiency can be expected from both structures.

5.3.2 PAPROS

The PArallel PROcessing System, (PAPROS), illustrated in figure 5.3, was described by Conrad and Wallach (1977) who were concerned with the solution of sets of linear equations. Like the structures described by Fong a single interconnecting bus is used. However, memories rather then processors are connected. A special processor, the controller, coordinates operation and has access to all memories. To enable access by both the local and controlling processors these memories are dual-ported.

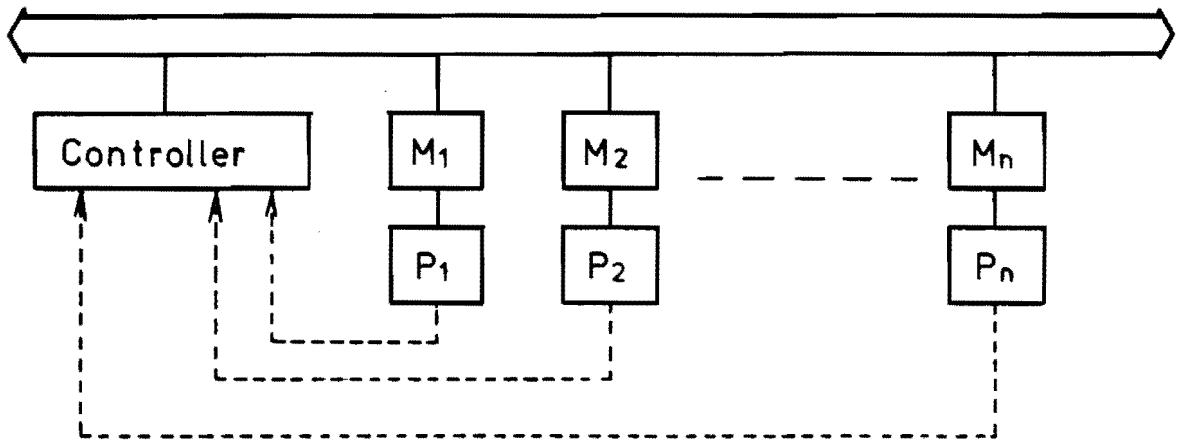


Figure 5.3: PAPROS System Structure

Execution on the PAPROS system consists of an alternating sequence of parallel and serial operations. The controller initially provides tasks serially for each processor. All processors then execute the assigned tasks in parallel and inform the controlling processor upon completion. When all tasks are executed the controller redistributes appropriate data and provides the subsequent set of tasks, and so on. The method completely avoids bus conflict, but two areas of inefficiency arise because of this mode of operation. The first region of inefficiency, defined here as forced serial stepping, exists because the controller operates serially leaving all other processors inactive. The second problem arises because the individual processors are unlikely to complete their tasks simultaneously. Hence, all but the last to finish will spend some period idling even during the parallel step. This will be called a global synchronisation problem.

Like the structures described by Fong, generality of allocation of tasks is not limited and a simple homogeneous bus structure is used. Hence, similarly, the structure is only suited to small systems even if the global synchronisation and forced serial stepping problems are insignificant.

5.3.3 MOPPS

The Multibus* Oriented Parallel Processing System, (MOPPS), (Shimor and Wallach, 1978) is similar in operation to the PAPROS system. As shown in figure 5.4 a controlling processor accesses the local memories of a number of processing elements. The serial-parallel nature of execution results in the same problems of global synchronisation and forced serial stepping.

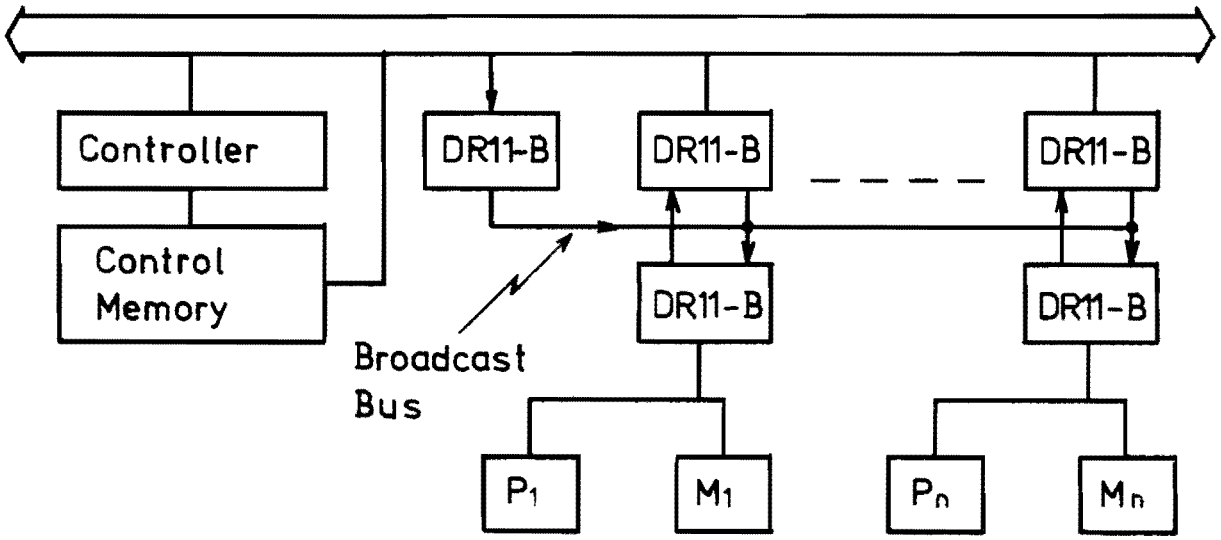


Figure 5.4: MOPPS System Bus Structure

Two points of special interest are apparent in the MOPPS structure. Firstly, a technique called back-to-back DMA is employed to transfer data between buses, and secondly, special hardware is included to enable data broadcasting to all processing elements.

DMA hardware is used to transfer data between devices such as disc storage units and memory. That is, they provide addresses and control

---

* - not INTEL's bus standard

signals as instructed beforehand by a supervising processor. The MOPPS system uses DR11-B* DMA devices in sets of two. One addresses the source bus, reading data which it passes to the second device which, in turn, writes the information to a second bus. The DR11-B can transfer bidirectionally so the sets of two form a bidirectional inter-bus data path. The rate of data transfer is close to that which could be achieved in normal transfers over the slower of the two buses. Because they are not involved, except for initialisation, processors could continue useful execution during transfers.

As seen in figure 5.4 an additional DR11-B is included specially to allow data broadcasting. It operates unidirectionally and provides data between each pair of DR11-B's while that member of the pair attached to the interconnecting bus is disabled.

5.3.4 DEMOS

The DEMOS 86 Multiprocessor (Brinkman et al, 1980) is a commercial system built with design criteria giving higher priority to reliability than is necessary in research oriented systems. In addition, considerable effort has been applied to ensuring software development is an easy extension from serial programming.

Like the MOPPS multiprocessor all of the system's memory is not globally addressable by every processor. Instead, as shown in figure 5.5, devices which can exchange data, and have direct access to each processing element's memory, are employed. These group bus interfaces (GBI) behave in pairs like back-to-back DMA controllers. The physical paths along which

---

* - DMA controllers which reside on UNIBUS (DEC(b), 1978)

data passes are called group buses. The system uses two of these, giving both redundancy for reliability, and increased available bandwidth.

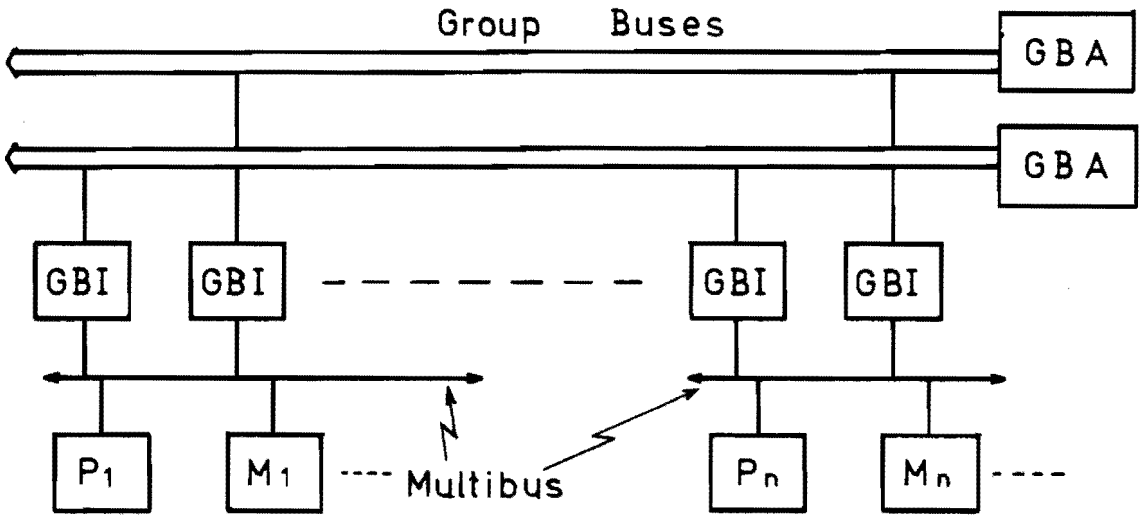


Figure 5.5: DEMOS System Bus Structure

Because many group bus interfaces are connected via a single data path, contention will arise. If multiple requests occur the group bus arbiters (GBA) decide which processing element will be given use of the group bus. As two parallel buses are used the level of conflict will be relieved to some extent. Broadcasting may be possible if more than one group bus interface can receive simultaneously.

5.3.5 CM*

The CM* system was built at Carnegie-Mellon University. It has been applied in the development of a variety of programs and many publications are based on its use. A recent paper (Deminet, 1982) describes the form of the system and a number of applications. CM* is the first system examined here which uses many interconnections between processors and memory units. A tree structure is used, as shown in figure 5.6, with a three level hierachy.

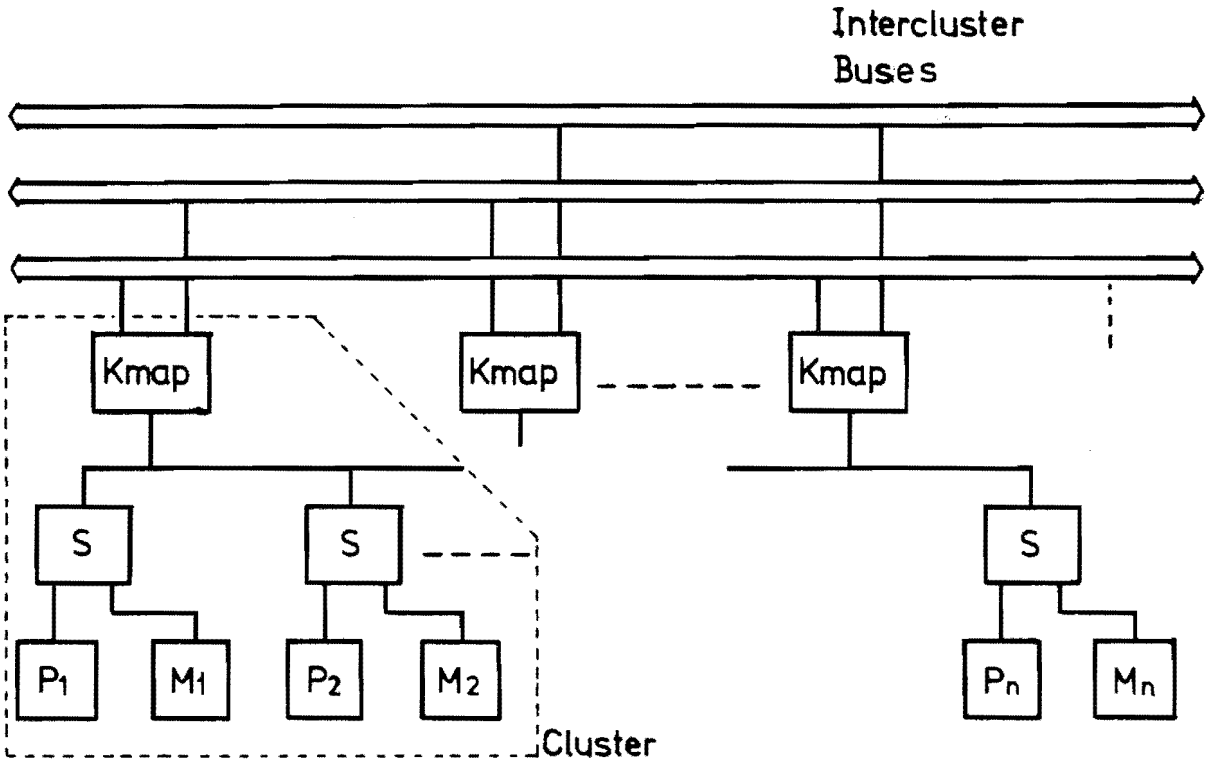


Figure 5.6: CM* Bus Structure

The lowest level consists of processing elements* connected via bus interfaces (labelled 'S' in figure 5.6) into clusters which share a single bus forming the second level. Between clusters a third set of 'intercluster' buses complete the hierachy. Memory mapping processors (labelled 'Kmap') connect the clusters to the intercluster buses.

All memory is addressable by each processor and, from a software point of view, accesses to all locations are identical. However, the number of interbus transfers required to physically access data results in a time cost ratio of 1:3:9 without considering contention ie. relating access times to (a) local memory, (b) memory within a cluster, and

---

* - LSI-11 single board computers (DEC(f), 1981)

(c) memory in other clusters.

The heterogeneous structure used in CM* means that a specific set of problems will be more suited to execution. Ideal problems would be those which are efficiently separable into cluster sized, almost independent tasks where a high proportion of memory accesses are retained within clusters. This loss of generality of application is the result of a well considered trade off between degree of specialisation, bus complexity, and the level of bus contention. The system is presently implemented using 50 processors and runs with acceptable performance degradation due to contention for a variety of applications (Deminet, 1982).

5.3.6 C.mmp

The development of the C.mmp multiprocessor, also at Carnegie-Mellon University, preceded that of CM* and, no doubt, provided ideas for a suitable structure for CM*. C.mmp utilises a full crossbar switching network as illustrated in figure 5.7. Consequently, the time cost of access to any element of any memory is fixed. All memories can be accessed simultaneously by any combinition of processors.

The C.mmp system uses 16 processors and 16 memory units. The size of the switching network needed for significantly larger systems becomes unfeasible as shown in the following section. The heterogeneous tree structure used in CM* is one solution to this problem, but homogeneous solutions are also possible.

5.3.7 FMP

The Flow Model Processor, (FMP), (Lundstrom, 1980) is of interest because it is aimed at efficient processor utilisation for very large numbers of processors ie. 512 processors sharing 512 memory units. The

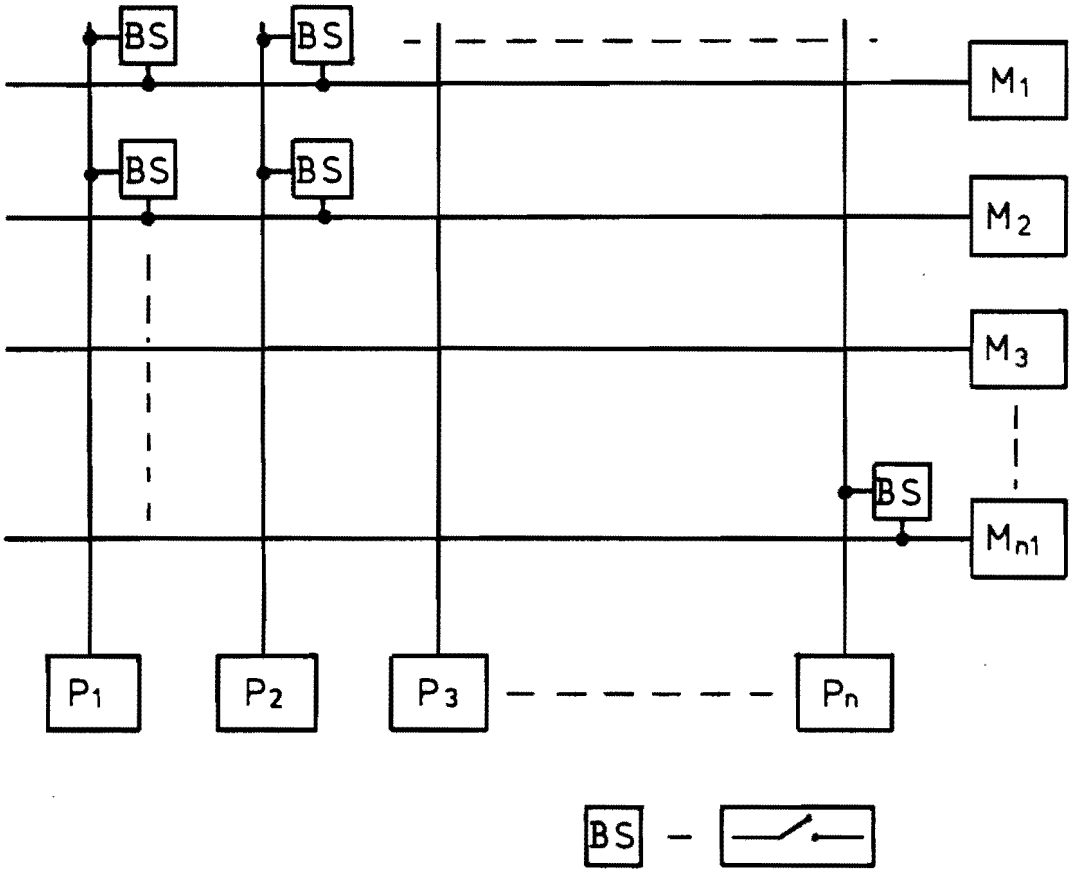


Figure 5.7: C.mmp Bus Structure

constraints of interconnection cost versus efficiency are, therefore, paramount.

Figure 5.8 illustrates the structure selected in the case of eight processors. It allows homogeneous access by all processors to every memory unit. The array is based on the OMEGA network described by D.H. Lawrie (1975). Topologically, the OMEGA network is equivalent to a number of others (Adams and Seigel, 1982) and, for example, it appears in figure 5.1 as the k-CUBE. The function of the bus switches differs from that used in the previous structures. To enable any processor access to any memory, the switches must be able to assume either of the states shown in figure 5.8 as full and dotted lines. In the paper Lawrie suggests that any processor could write to all memories simultaneously if the broadcast switch

positions (figure 5.9) are included.

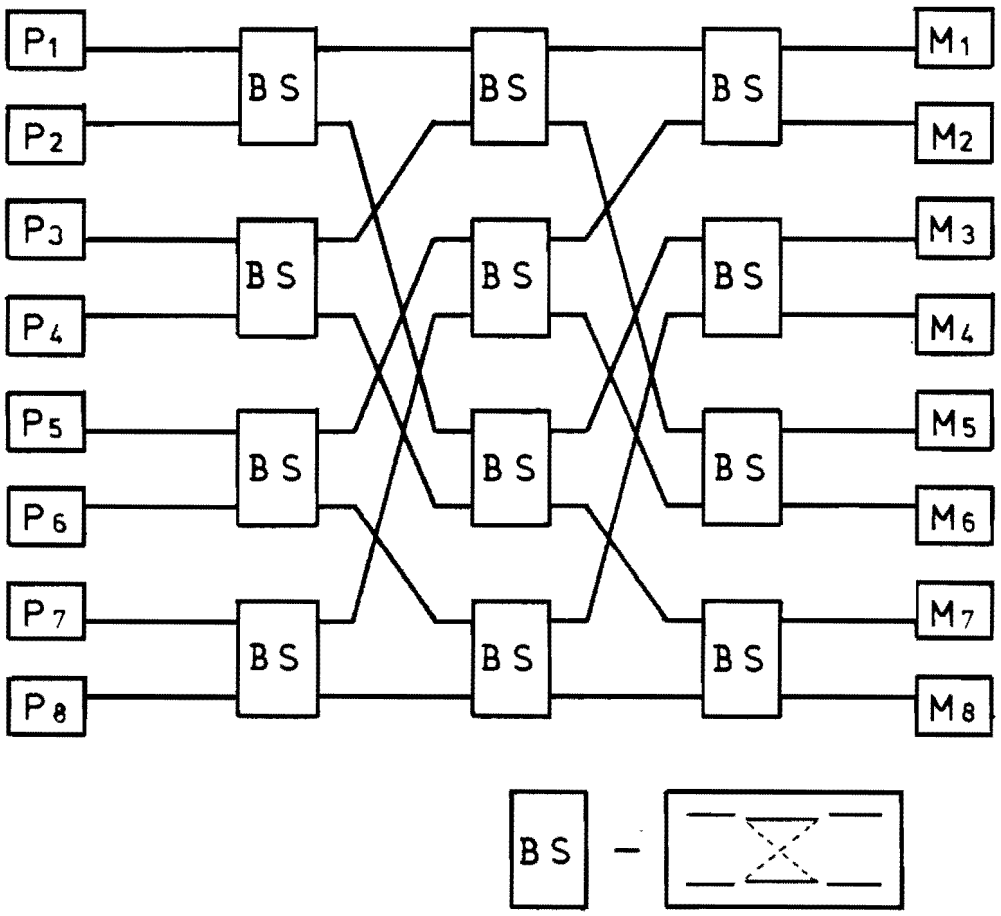


Figure 5.8: FMP System Bus Structure

Figure 5.9: OMEGA Network Bus Switch

The bus switches can be constructed using four of the simple switches employed, for instance, in C.mmp. Comparing the OMEGA network

with the full crossbar, and assuming the number of processors equals the number of memory units, consider an eight processor system:

.full crossbar requires 8x8 = 64 simple switches

.OMEGA requires 3x4 = 12 OMEGA switches = 48 simple switches

Although the difference in requirements is small the OMEGA network has a size in proportion to nxln(n)*, whereas the full crossbar increases in proportion to nxn, where n is the number of processors which also equals the number of memory units. Consequently, when n=512 the OMEGA network requires 9,216 simple switches which is only 3.5% of the 262,144 needed for a full crossbar.

Control strategies for switches within such arrays have not been considered. The full crossbar is relatively easy to control as individual switches are only concerned with a single processor and a single memory unit. Hence, control for each switch requires the decoding of requests within a fixed address space. Other networks, such as OMEGA, require more involved control as free data paths through shared switches must be established. For the OMEGA network the paths are unique. Other networks may allow many optional paths from which a selection must be made.

5.4 Malfunction Detection and Recovery

A strong impetus for the implementation of multiprocessors, as mentioned in Chapter 3, is the reliability achievable using the inherent redundancy of parallel processing elements. This redundancy can only be exploited if malfunctions can be detected and subsequent recovery

---

* - ln = logarithm to base 2

accomplished. These needs are reflected in special hardware implemented in existing non-highly parallel multiprocessors. For instance, the UNIVAC 1108 utilises a count down timer which will interrupt operation unless it is regularly informed of satisfactory operation (Enslow, 1974, p.73). An independent processing entity, the Availability Control Unit (ACU), automatically initiates a recovery sequence on receipt of such an interrupt. Note that no information about the state at the instant of malfunction is recorded.

In a research oriented environment recovery is not important. Instead, the detection of faults and retention of the state at the time faults occur is desirable, giving a user information allowing reconstruction of the events leading to the fault. The IBM System 370 uses a Malfunction Alert interrupt which can inform all processors at the instant of a fault (Satyanarayanan, 1980, p.155).

As an aid to either recovery from or reconstruction of a fault situation, a valuable facility would be one which impeded access by individual processors to common resources. After detection, tests using such a facility could isolate offending processors. The Burroughs B7700 and C.mmp have registers associated with each memory unit with processor-mapped bits which, when set, stop access by respective processors (Satyanarayanan, 1980, p.155).

5.5 Salient Features of Existing Systems

The preceding sections have described a number of features of existing multiprocessor systems. These are now briefly reviewed using, for perspective, the structural needs of the UCMP system considered later but summarised here:

1. The system must be cost effective implying small size but allowing extrapolation to gauge the expected performance of large systems in the execution of transient stability analysis programs.

2. If possible, scope to examine the limitations and capabilities of the selected bus structure should be included.

3. Hardware support for improved operating system efficiency, where cost effective, is desirable.

The following are observations and features related to these requirements:

- The bus structure should be appropriate to the size of the system. Complex networks, such as FMP, utilise highly interconnected structures while small systems, eg. PAPROS, may use a single bus.

- Bus switches are difficult to implement because (a) they represent complex hardware, and (b) a control strategy must be considered and consequent additional hardware included.

- Hardware implied global synchronisation and forced serial stepping, as in MOPPS and PAPROS, are undesirable.

- Inter-processor, rather than processor-memory, interfaces require the cooperation of two or more processors and, hence, (a) require control signals to interrupt processors during exchanges, and (b) delay at least one processor unnecessarily.

- Heterogeneous structures imply non-uniform processor effectiveness because physical position affects performance. A trade off between ability to be used in a general purpose manner

and efficiency, at fixed cost, is applied in deriving structures, such as CM*.

- Heterogeneous structures require different programs for each processor, or set of processors, that have a different view of the system. Hence, software development is more complicated than for homogeneous systems where all or most processors can execute identical code.

- Computer operating systems can be enhanced by special hardware features. For multiprocessors a number of processing elements may require identical program code which, at run time, resides symmetrically in a number of memory units. A broadcast capability allows writing of identical information to many memory units and, as such, is ideally suited to enhancement of program loading. In addition, the inclusion of broadcasting facilities broadens the options of the programmer.

- Dual-porting allows access to the local memory of a processor without the cooperation of that processor. Hence, serial operating system functions, such as program loading, can be simplified by involving only a single processor.

For connected systems with more than one bus, various methods can be employed in the inter-bus transfer of data. From the examples cited four distinct methods are identified:

- two processors cooperate. One presents data, for instance on an I/O port, which can be read by the other.

- direct inter-bus memory mapping. Memory on one bus is directly accessible to processors on the other. (eg. as used in

CM*)

- back-to-back DMA. Two non-processor devices, one on each bus, exchange data between locations on both buses as instructed by processors. (eg. MOPPS, DEMOS)

- dual-port memory. Special memory exists in the address spaces of two buses and, hence, can be used either as a stepping point for transfers, or as common memory.

CHAPTER 6

HARDWARE REQUIREMENTS AND IMPLEMENTATION OF THE UCMP MULTIPROCESSOR

6.1 Introduction

The execution performance of multiprocessing algorithms can be measured most accurately and most confidently during execution on real parallel processing hardware. This chapter outlines the requirements and implementation of a multiprocessor, designated the UCMP system, built to enable such testing. Many of the valued features of existing systems identified in Chapter 5 are employed. The measured performance of this hardware can be combined with performance estimates produced by the simulator described in Chapter 8 to give a reliable indication of the modelled algorithm's execution characteristics over a wide range of processor number and input conditions.

Requirements of the UCMP system were determined using the UCTS program as a benchmark. Data storage needs were based on the largest invisaged power system networks to be modelled. Where possible considerable margins are allowed for unforseen expansion. Although the system was designed around a benchmark, its structure is not highly specialised and is homogeneous. It, therefore, could be used to evaluate the execution characteristics of a wide range of algorithms.

Two configurations, each giving precedence to different priorities, are discussed. As it is expected that, in a research environment, program development and debugging would be frequently required, features aimed at easing these tasks have been given high priority in one configuration. The

alternative configuration permits high throughput during tests involving varying numbers of processors and different sets of code and data.

Available equipment has been used where possible. In particular, microprocessor development facilities and a departmental minicomputer are employed. An interface to the host computer permits a choice of data paths and very high rates of data transfer. In addition, it is designed to operate with minimal interference to multiprocessing operation. A small proportion of the hardware was purchased at the board level while most items were developed using discrete components through a series of projects (LOW, 1980; BARTH, 1981a; BARTH, 1981c; PARR, 1981; BAILEY, 1981). Most of the items developed are now commercially replaceable. Two notable exceptions are a host processor interface board and special debugging aids.

As a result of the processor selected as the basis for each processing element, raw processing power is relatively low at around 10,000 FLOPS. This figure is of little consequence, however, in the invisaged research oriented application where the relative speed of various numbers of processors is the performance measure. Note must be taken, though, that extremely slow processors can bias results by ensuring low bus utilisation and, hence, masking potential conflict problems. During development, new processing devices which are compatible with the processors employed, became available and could raise processing speeds by a factor of 100. Such speeds would place the system in the same class as many very expensive existing scientific mainframes at around a million FLOPS raw processing speed.

Hardware within the UCMP system allows measurement of performance. In addition, provision is made to apply deliberate constraints to facits of operation enabling practical evaluation of sensitivity to these factors. For instance, the degree to which bus usage approaches saturation can be

varied by altering the speed at which the bus operates.

6.2 Functional Requirements, Options, and Components Selected

The hardware needs of the UCMP system can be categorised as firstly, a multiprocessing capability, and secondly, support facilities enabling code preparation, execution monitoring etc..

A fundamental design choice is the selection of a suitable number of processors. Factors influencing selection include cost effectiveness, the complexity of implied bus structures, and the degree to which the system is capable of realistic performance, indicative of that expected with larger practical systems.

Experience with respect to program length, and the format and quantity of data required to describe power systems, leads to a determination of the amount of memory necessary. In turn, the range of suitable processors is narrowed by the need to address this memory.

Invisaged applications of the UCMP system require a separate execution facility for those sections of programs where parallel processing is of little interest. Combined with this, facilities for code production, testing, and storage are needed. Accurate evaluation of performance using suitable timing elements is also necessary.

A number of manufacturers produce processors and support buses enabling multiprocessing on a suitable scale. For economic reasons the Department of Electrical Engineering has restricted investment to INTEL equipment and development systems. The department also has a Digital Equipment Corporation minicomputer (VAX 11/780). Hence, there is a strong impetus to select components compatible with the equipment provided by these two manufacturers.

6.2.1 Numerical Representation

In digital systems, elements of continuous sets of numbers must be approximated by finite length bit strings. A trade-off between three parameters is made in creating such representations:

.the length of the string,

.the range of values represented, and

.the accuracy with which individual values are represented.

In addition, the accuracy in sections of the range can be improved at the expense of accuracy in others.

In practical applications string representations of continuous sets of numbers are defined by applying organised significance to each bit and groups of bits. Two common representations are fixed and floating point. Fixed point numbers have unity spacing between subsequent values. As such, in digital systems, they offer exact definition of integers over a finite range. In indexing and counting functions fixed point representation is therefore valuable. They have constant absolute* error over the full range they represent.

In floating point representations bits within strings are organised in two fields: one, similar to fixed point representation, called the mantissa, and another called the exponent which sets the range within which the mantissa lies. In representing real numbers floating point is most

---

* - absolute error refers to the difference between successive numbers while relative error relates this to the magnitude of the number represented.

commonly implemented and attention here is limited to its use. Floating point schemes have approximately constant relative error.

In this section formats used on commercially available machines are described and consideration is given to determination of the most suitable floating point format for transient stability analysis.

Floating point representations used in a number of computers are compared in figure 6.1. Both the range and error distribution for all schemes are illustrated. The measure of error is based simply on the difference between successive numbers. This, however, is not a perfect measure in comparing schemes because no account of rounding is taken.

6.2.1.1 Wordlength Required

An investigation has been made using the UCTS program with benchmark power system data to establish a suitable floating point wordlength (Saunders, 1979). This study is now described and the results are shown to confirm the experience of other researchers (Happ et al, 1979).

Quantitative measurements were made using the UCTS program on a Burroughs B6700 computer. This uses 48 bit words divided into a 39 bit mantissa, an 8 bit exponent (base 2), and sign. The method of testing was simply to successively reduce the wordlength (mantissa only) until the results of the program exhibited significant deviations. Two sets of data were used as benchmarks; one a very simple single generator system, and the other a 23 bus, 5 generator system. Figure 6.2 illustrates the performance of the program for mantissa lengths between 21 and 39 bits. It was found that below 21 bits convergence could not be achieved.

Significant introduction of error occurs between mantissa lengths of approximately 21 and 25 bits. Happ et al (1979) suggested that array

	Wordlength	Exponent length	Base
A	64	11	2
B	48	8	2
C	38	8	2
D	32	8	2
E	32	2	2

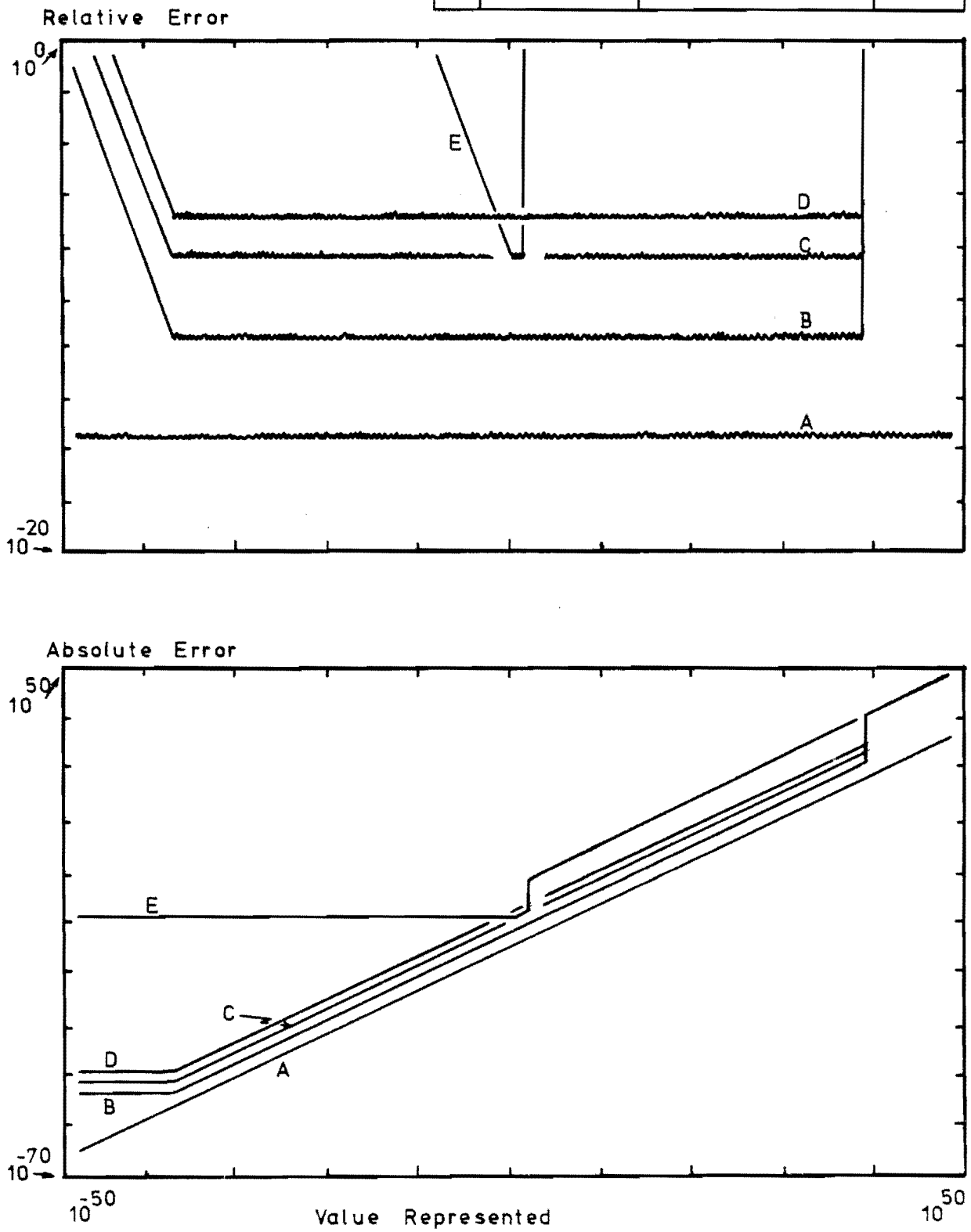


Figure 6.1: Comparison of Five Floating Point Representations

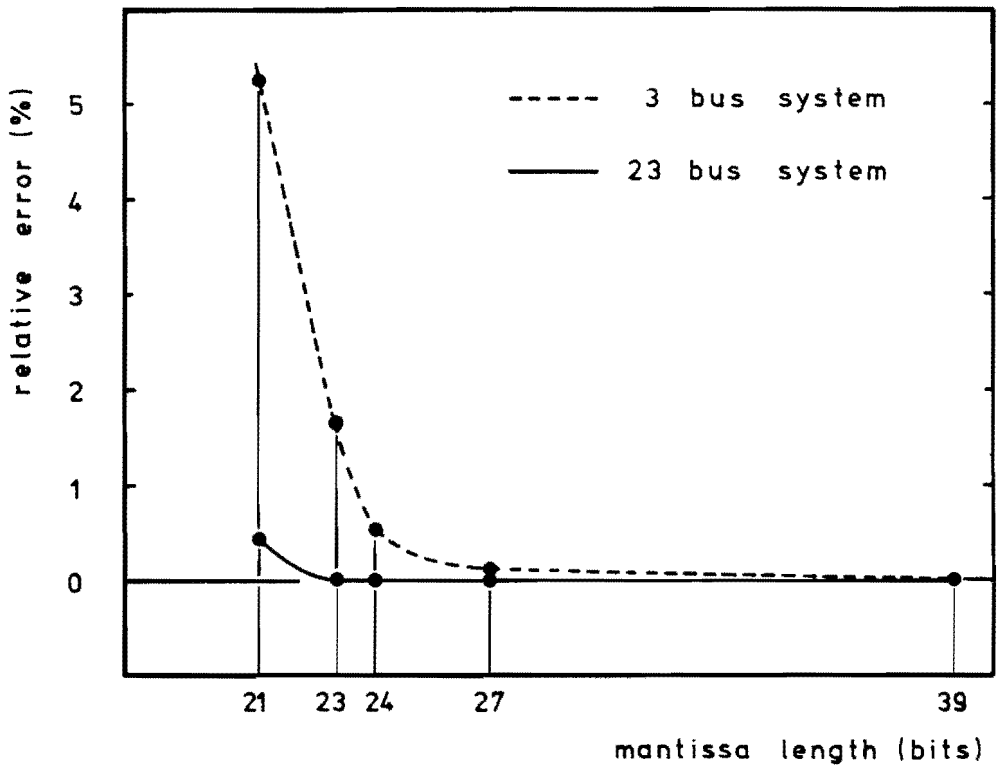


Figure 6.2: Errors in UCTS Program Output Resulting from Reduced Mantissa Lengths

processors with 32-bit floating point representations, ie. 23 bit mantissas, are not sufficiently precise, but that 38 bit machines, such as the Floating Point Systems AP120-B ie. with 29 bit mantissas, are satisfactory.

Aided by the use of per unit quantities, transient stability analysis programs require only a limited range of numbers with accurate representation. An eight bit exponent with a base of two provides far more range than is necessary. Hence, a solution to maintenance of both high accuracy and low wordlength may be to sacrifice bits in the exponent to extend the mantissa. Figure 6.3 illustrates this rearrangement using a 2-bit exponent with a base of 16. This scheme would satisfy the known accuracy requirements, but unfortunately is not commercially available.

	Wordlength	Exponent length	Base
A	32	8	2
B	32	2	16

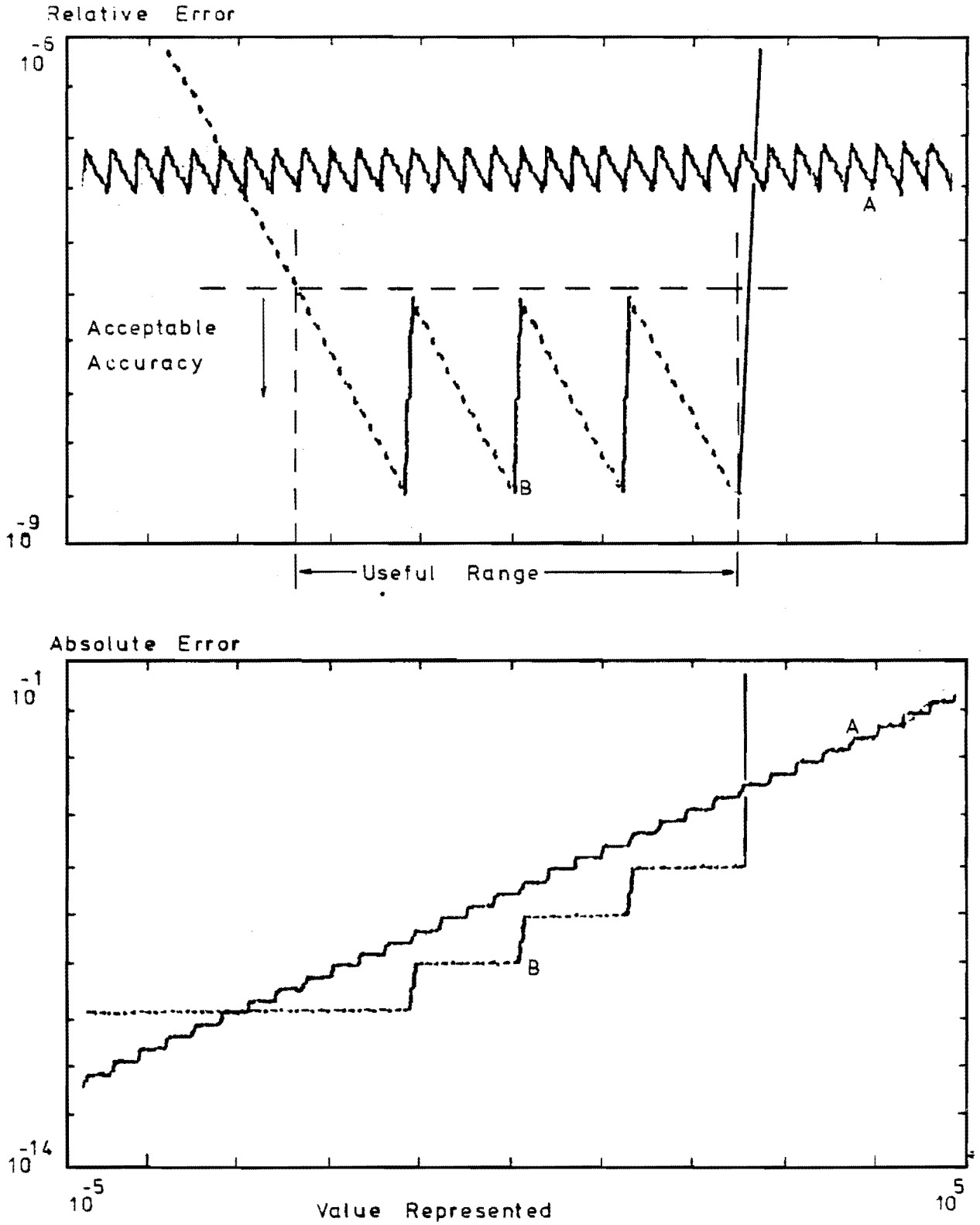


Figure 6.3: An Adequate 32-bit Representation

Floating point representation using 32-bit word lengths is a very common standard available on many mainframe and mini-computers which are often based on 32-bit architectures. Microprocessors with 16-bit data handling also tend to use 32-bit format ie. two word. Extended precision (typically 64-bits) is available on many machines. This more than adequately satisfies the requirements for accuracy, but increased execution times can be a problem.

6.2.2 Memory

Two categories of physical memory are assumed:

.local memory, accessible to processors without use of a shared bus, and

.globally accessible memory, which can be written to and read from by all processors.

Three types of information which must be stored are categorised as follows :-

.Definitely Global - data which must be available to all processors.

.Preferably Global - data which would most conveniently be available to all processors. If this information is not globally available then either it must be stored a number of times elsewhere or individual processors must be assigned specific tasks before execution commences.

.Local - information such as program code, intermediate results of operations, and stack contents, which need only be available to individual processors.

For a specific hardware implementation the choice between storing preferably global data in locally accessible or globally accessible memories will depend on any bus contention problems involved when using globally accessible memory.

Benchmarks selected to determine quantities of memory are necessarily very close to the invisaged largest cases required for transient stability analysis.

6.2.2.1 Global

Local storage of preferably global data implies either redundancy of storage or undesirable predefinition of task allocation. Therefore, preferably global data will be assumed to be stored with global accessibility. Data to be stored includes management related objects, such as semaphores, a static description of a power system, and the state of that power system as it varies during program execution.

Power systems modelled by other researchers include some very large networks. For instance, F.M. Brasch (Brasch et al, 1981) used systems with approximately 1700 buses and 400 generators. However, smaller networks provide more challenging problems for multiprocessors because parallelism is reduced by the availability of a smaller number of tasks. Using 32-bit representation of real numbers, and 16-bit integers, 128K bytes of memory can contain all data necessary to model a 1000 bus network with 99% sparsity coefficient, and also 200 generators each with the most detailed model available. Boards with 128K bytes of memory are readily available. Although this is considered more than sufficient, expansion to larger systems could easily be implemented using more or larger memory units.

6.2.2.2 Local

Information stored locally by each processing element includes; program code, data (intermediate results during execution), and stack contents. Accurate assessment of the likely storage in these categories was made using programs which were intended as the complete transient stability analysis implementation. The executable code* translated from these programs totalled approximately 20K bytes. Although this code was untested at the time of hardware selection, final memory requirements would be very unlikely to differ markedly. Matching this requirement with commonly available components 32K bytes of memory is the most appropriate realizable level. Considerable margin is left, therefore, for additional program code if necessary.

6.2.3 Number of Processors and Bus Structure

Effective utilisation of hundreds of processors is possible in the analysis of large power systems. However, the cost and complexity of such multiprocessors is restrictive, and unnecessary, in a research environment ie. where validation and comparison of algorithms is the prime objective. When small power systems are analysed limits to the use of multiprocessors are measureable with only a few processing elements. Hence, using such systems, algorithms can be evaluated on a realistic basis. In addition, the performance of serial simulations, modelling the execution of multiprocessors, can be validated using small power system descriptions. The same simulations can then be used with greater confidence to assess the likely performance of systems with many more processing elements.

---

* - 8086 instruction set

In Chapter 5 the strong relationship between the complexity of bus structure and the number of processing elements was illustrated. Because of this relationship, selection of the most cost effective number of processors necessitates a simultaneous consideration of appropriate accompanying bus structures.

DEC produce a range of single board computers, called LSI-11's, suited to application as processing elements in the formation of a multiprocessor. CM* (Deminet, 1982) uses them. The LSI-11 bus (also called Q-Bus and sub-UNIBUS (DEC(a), 1979)) allows a maximum of a quarter of a megabyte of directly addressible memory, and has limited capabilities with respect to multiprocessing ie. signals oriented towards use with DMA devices can be manipulated to allow the inclusion of second and further processors. Because of this limited structure, CM* uses specially developed hardware to interface the processing elements to cluster and intercluster buses. Another bus standard supported by DEC is UNIBUS which is standard for PDP-11 processors, and is available on VAX computer systems. UNIBUS is also constrained to an 18 bit address which is inadequate for transient stability analysis, and has weak multiprocessing capability.

INTEL, on the other hand, produce a wide range of single board computers configured to use a bus, called MULTIBUS (INTEL(d), 1978), which is designed to handle multiprocessing. Although it was created by INTEL, MULTIBUS is now used extensively by other manufacturers. At one megabyte, the address space is sufficient for transient stability analysis. Using a serial priority resolution circuit up to 16 processors can assume the position of bus master.

Cost limitations affect the maximum number of processing elements. Fortunately, when establishing the effectiveness of an algorithm, the

marginal value of additional processors declines rapidly. Early experience with the simulator described in Chapter 8 suggested that small power systems provide challenging difficulties for multiprocessors with fewer than 10 elements even when using elemental linear solution techniques. Combining the economic restraints with practical value, therefore, the probability that more than 16 processing elements would be included is low. Hence, MULTIBUS appears to offer a suitable hardware basis for multiprocessing. Before accepting it, however, the degree to which bus contention degrades performance must be shown to be acceptable.

Difficulties arising in modelling bus conflict are considered in Chapter 9. A simplistic approach is adequate here as wide margins for error clearly emerge. During execution of the transient stability analysis program the worst case point, ie. where the hardware connection to global memory will be busiest, is during the linear substitution steps. This program section is therefore selected as a suitable benchmark to assess the level of bus traffic. The processor operating speeds used to assess the access rate to local and globally accessible memories are those of the 8086 processor operating at 5MHz as shown in figure 6.1. Measured speeds differed significantly from the execution rates described by the suppliers (INTEL(h), 1980). The figures were based on execution rates selected from the available information to ensure, if anything, a pessimistic estimate of bus utilisation and performance.

Individual processing elements can contain local memory useful for storage of code and data. In a system with no such memory all accesses require use of MULTIBUS. Hence, as a single element uses virtually all of the available bandwidth, second and further processors are of little value. The use of local memory is therefore essential when using such a single time-shared bus.

For a typical update step in the linear substitution process, consisting of a complex multiplication and addition, four 32-bit real values are read from global memory and two are written. Also at least three accesses are made to set and reset semaphores and for each task two index values are read. Using 16-bit words and allowing for a few extra semaphore checking accesses, about 20 transfers via the single bus will be required ie. 12 μsec given a 600 nsec typical access time. From table 6.1, the total execution time will be approximately 1400 μsec. Therefore, a figure of about 1% total bus utilisation per processor may be expected.

OPERATION	8086 (μsec)	B6700 (μsec)
substitution of diagonal elements	1172	44
substitution of off-diagonal elements	1400	76
search each element of Dynin	20	16
semaphore set and reset	18	10
decrement element of Dynin	10	6

Table 6.1: Execution Times Used to Model the Operation of the 8086 and B6700 Processors

Using 10 processors with 1% bus utilisation each, the possibility of a bus conflict at each access is close to 10%. Therefore, an increase of about 10% in average access time to global memory is expected and, as this represents approximately 1% of all accesses, a loss in performance by each processor, and of the system as a whole, of about 0.1% is likely. Extending this towards 100 processors bus contention overhead would increase to about 10% as, not only is the bus more likely to be busy, but

also the number of accesses pending from other processors, on average, is greater. For more than 100 processors bus saturation, ie. 100% bus utilisation, will occur rendering any further increases in the number of processors useless.

Up to 16 processors with local memories can, therefore, comfortably utilise MULTIBUS.

If local memories were dual-ported, further flexibility would be offered to programmers. Dual-porting allows access by all processors to the local memories of all the other processors. It thus provides programmers with further flexibility which could enable more useful processor utilisation. Also, dual-porting is invaluable in simplifying the operating system load function because no local processor cooperation is necessary when access to all local memories is required.

6.2.4 Processor Capability

At the time of design of the UCMP system, two INTEL processors, the 8085 and the 8086, offered suitable capabilities. Of these, only the 8086 has sufficient address space to cover the needs of transient stability analysis. Apart from enhanced addressing, the 8086 has features and support devices which clearly identify it as an appropriate choice for the UCMP system.

6.2.4.1 Addressing

Combining the conclusions of the previous two sections, a total memory requirement of approximately 500K bytes is expected. Assuming that local memories are dual-ported all of this memory must be addressable from MULTIBUS and, in turn, by individual processors. Single board computers, based on the 8085 processor, can address and utilise a full megabyte

address space although the 8085 can only directly address 64K bytes eg. the AM95/4010 Monoboard Computer (AMC, 1980). This expanded addressing is paid for in software by additional code requirements and time being spent in execution of paging functions, and in hardware by increased complexity. Consequently, use of a processor which can address the full necessary space is desirable. Although the 8086 can address one megabyte, processors made by other manufacturers have larger address spaces and, therefore, allow greater margin for expansion eg. the Motorola 68000 (MOT, 1981) can address 16 megabytes. The instruction set of the 8086 is oriented towards 64K byte pages. Although this has no effect on hardware, it introduces minor limitations in high level language implementations.

6.2.4.2 Program Functions

The transient stability analysis program requires data formats as follows:

.integers with a range of a few thousand, and

.real values with at least, and preferably more than, 32-bit floating point accuracy.

These formats are supported by software available for the 8086. However, because of its 16 bit structure, and slow execution of multiplication and division instructions, the speed achieved in floating point operations is poor.

The architecture of the 8086 is arranged to enable cooperation between itself and a coprocessor. A number of coprocessors are available and, of these, the 8087 Numeric Data Processor (INTEL(a), 1982) provides hardware enhanced fixed and floating point operation. As illustrated in table 6.2, typical speed improvements can be in the order of 100. (These

figures are quoted from references INTEL(h) (1980) and BURROUGHS (1975)). In addition, longer wordlengths are available. As 32-bit representation is likely to be inadequate, inclusion of a capability to handle more if necessary is valuable, leaving options open to the programmer.

Processor/ Accuracy	Addition (μsec)	Multiplication (μsec)
8086 (32 bits)	1600	1600
8087 (32 bits)	14	18
8087 (64 bits)	14	27
B6700 (48 bits)	8	11

Table 6.2: Comparison of Basic Operation Execution Times

In situations where more than one processor can assynchronously update the contents of critical common memory a feature called bus locking is required. This enables a processor to obtain unhindered use of the bus for consecutive bus cycles. During such sequences semaphores can be set securely. The 8086 supports bus locking and, hence, can set semaphores quickly, and requires a minimum of hardware to do so.

6.2.5 Support

Components considered so far fulfill the execution needs of the UCMP system. To support these in practice further hardware is required to enable the following:

.program development and debugging,

.storage of large blocks of code and data,

.separate serial execution of program sections,

.interfacing to a user,

.loading code and control of execution, and

.performance evaluation.

Available tools, with features suited to the satisfaction of these requirements, are the VAX 11/780 computer and the INTEL Microcomputer Development Systems (MDSs). In addition, closely associated with the core elements, a number of items of hardware for interfacing to these tools, and to the user, are necessary.

Two stages are invisaged in use of the UCMP system. The first involves the development of reliable code using the code preparation and debugging facilities of the MDS. Emphasis here is on effective debugging with little need for speed. The second stage involves interactive execution under a variety of conditions, including changes of prepared code and differing numbers of processors, with speed and flexibility.

Consequently, two configurations of the UCMP system are proposed. The simpler requires interfacing to the MDS. This is a task for which the manufacturers provide a variety of options. At the time of conception of this project the VAX computer was employed in serial development and use of the UCTS program. It has both mass storage and user interfacing facilities. As such it offered a suitable system host assuming a satisfactory interface to the UCMP system. Hence, the second, and more complex configuration requires an interface between the VAX computer and the UCMP system.

To attain speed and consequent efficiency data must be transferred bidirectionally through the interface at a high rate enabling quick

replacement of code or data, possibly during execution. Used only as a loading facility transfer rates in the order of 2K bytes per second would allow the code for a single processor to be loaded in about 10 seconds which may well be considered acceptable. On the other hand, if transfers were expected during execution without introduction of excessive delays rates in the order of 100K bytes per second would be necessary ie. close to the expected MULTIBUS utilisation. Serial transfer methods would be inadequate in these circumstances. Therefore, the four parallel inter-bus transfer options identified in Chapter 5 were considered.

6.3 UCMP System Description

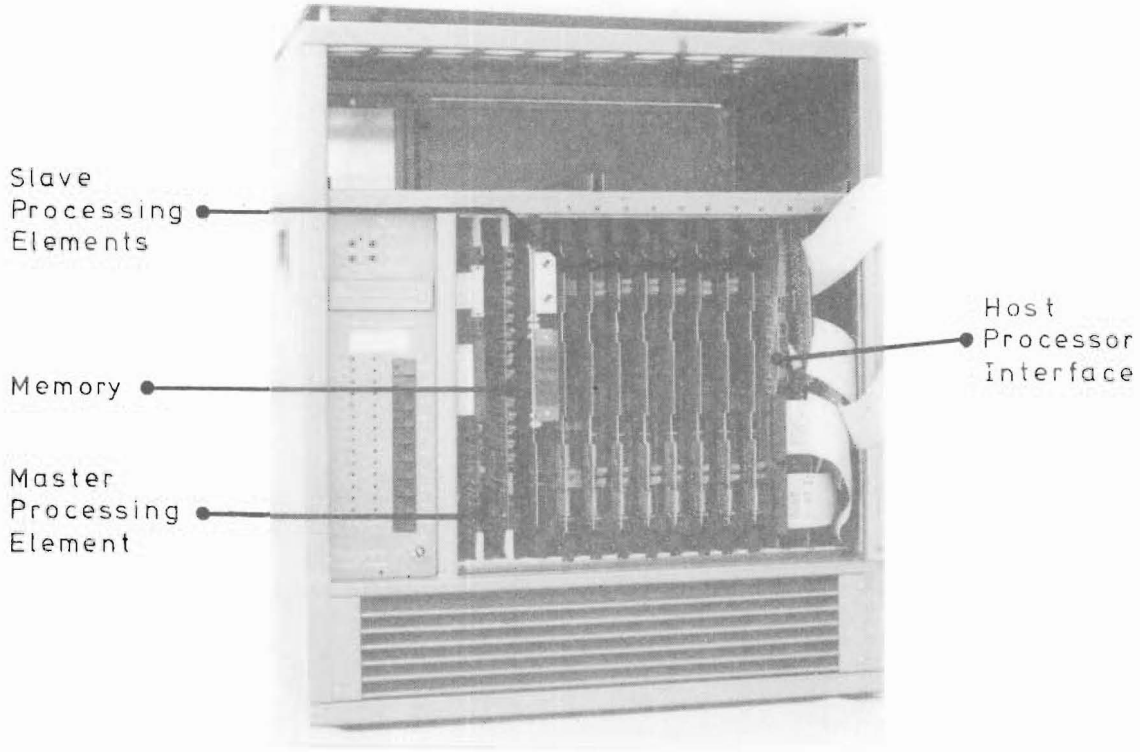
The core elements of the UCMP system (ie. those involved in execution at run time) are connected by a single bus. Up to 12 processing elements can be employed. Other devices directly connected to the bus include globally accessible memory and an interface to a host processor. Photograph 6.1 illustrates the chassis with processing elements, memory, and the host interface in place. A separate control bus coordinates operation of the processing elements.

6.3.1 Bus Structure

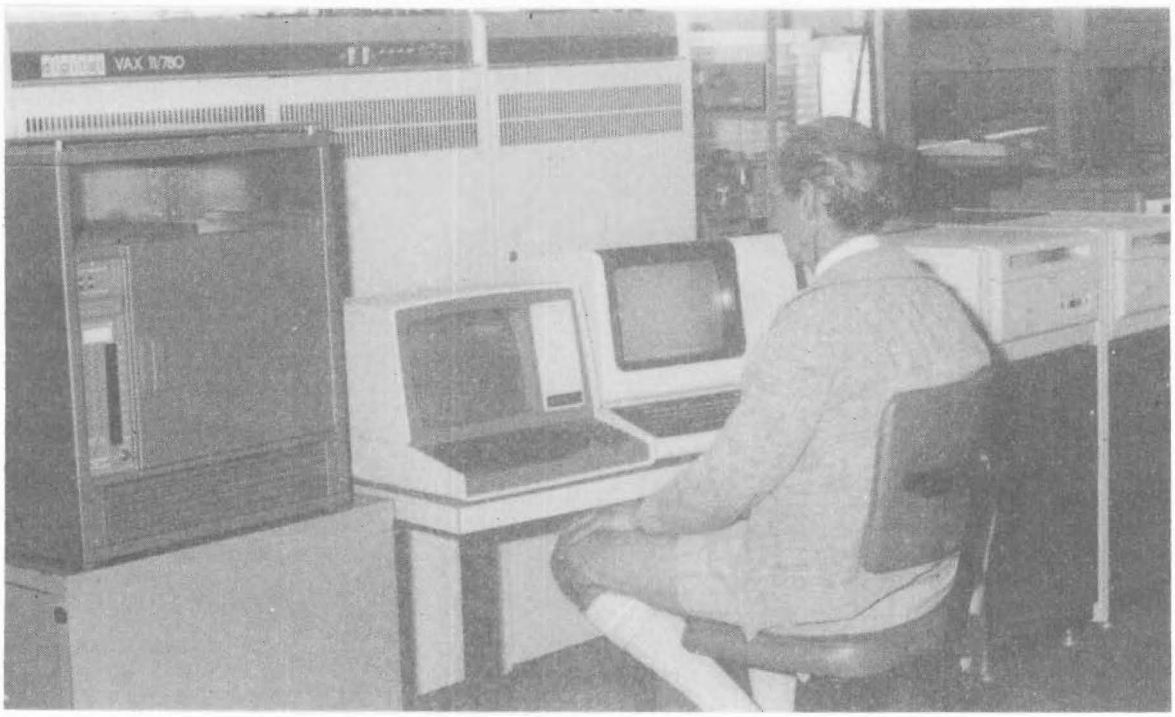
The single bus employed conforms to INTEL's MULTIBUS specification (INTEL(d), 1978). Globally accessible memory, processing elements, and a host processor interface are connected as illustrated in figure 6.4. Any data transferred between, or shared by, processing elements is accessed via MULTIBUS.

6.3.1.1 Processing Elements and Local Memories

Each processing element contains an 8086 processor and 32K bytes of dual-ported random access memory. 128K bytes of globally accessible memory



Photograph 6.1: UCMP System



Photograph 6.2: VAX Computer Based Operation

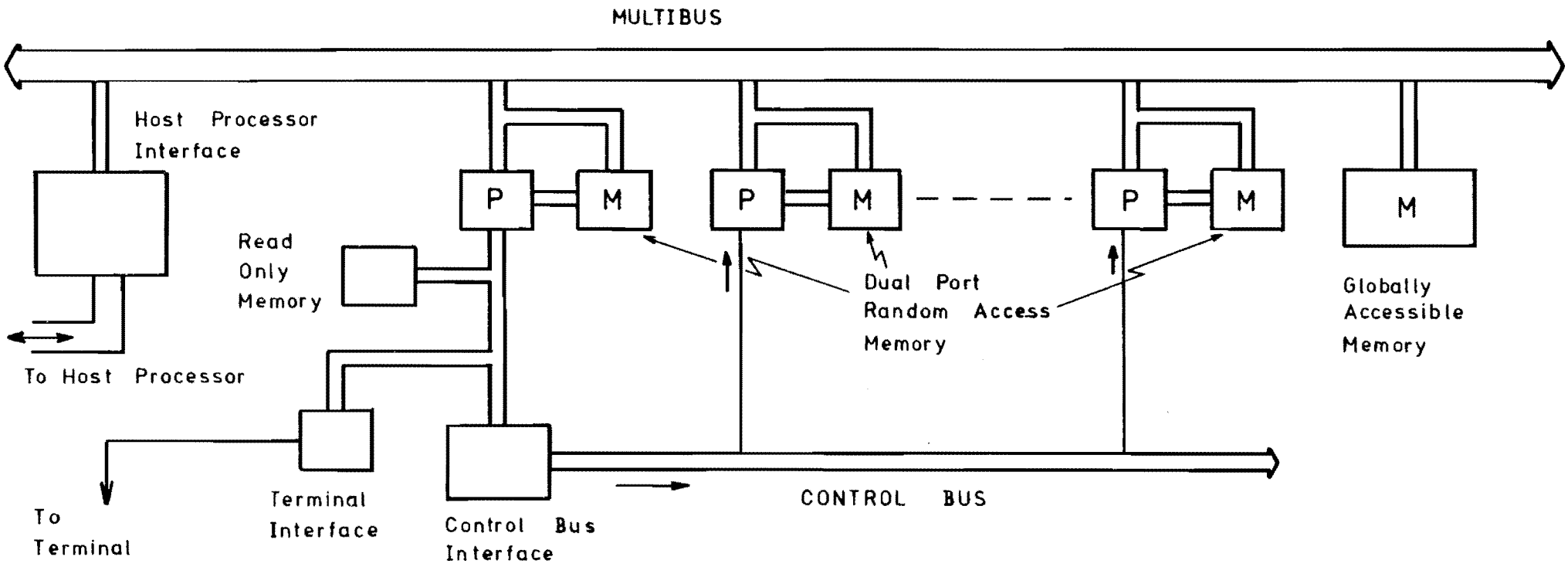


Figure 6.4: UCMP System Structure (core components)

provide more than expected necessary storage of this type. Provision is made, however, for the simple addition of further 128K byte blocks if necessary. The host processor interface has access to all of the MULTIBUS accessible memory.

To facilitate control, processors are arranged in a hierachical structure. This is not to say, however, that at run time task distribution is necessarily predefined. One processing element, designated the master, contains additional hardware to:

.control other processing elements,

.interface to a user, and

.permanently store a minimal operating system.

Via the control bus the master processing element has a supervisory capacity with respect to all the other (slave) processing elements.

Slave processing elements need no local read only memory. This represents a significant saving in hardware. Although, as a result, slave processors cannot operate independently, performance as a system is not degraded. Two factors contribute to this fortunate situation. Firstly, because local and MULTIBUS access to local memories need not use coincident addresses, a degree of freedom in addressing, discussed later, can be exploited. Secondly, the dual-port memories associated with each slave processor can be loaded while slave processors hibernate. The code loaded can then, when executed by the slave, establish ordered operation of the slave processing element.

Because dual-porting is employed, all processors can directly access the local memories of all other processing elements. As such these memories can be considered globally accessible. However, their use as such

is constrained and degraded as follows:

.When dual-port memory is accessed from the local processor and MULTIBUS simultaneously one or other of the requests is delayed. Therefore, speed is reduced.

.Semaphores cannot be stored in local memories. A local processor setting a semaphore in its own memory cannot lock MULTIBUS. Therefore, a second processor can interfere in the two cycle setting sequence corrupting the operation of the semaphore.

.Items stored in local memories are referred to by either of two distinct addresses ie. one by the local processor and another by the rest. Therefore, if local memories are referred to by other processors, software development is complicated and, unless task allocation is predefined, the quantity of code and execution times must increase.

6.3.1.2 Priority Resolution

When multiple simultaneous requests are made for MULTIBUS, hardware on the backplane arbitrates and selects a bus master. As implemented, priority among processing elements is fixed. This approach permits the maximum possible number of processing elements without reducing the operating speed of the bus. Alternative approaches can be advantageous in particular circumstances. For instance, a rotating priority method would be valuable when MULTIBUS is close to saturation. In Chapter 9 a study of the effects of saturation is limited by the implemented resolution scheme.

6.3.1.3 Malfunction Considerations

In Chapter 5 the requirements for action on the occurrence of an error were identified as:

.detection of the fault as soon as possible, and

.immediate recording of the state of the system.

Detection of faults is not simple during execution. Some criterion must be applied to observed changes from normal execution patterns to distinguish a fault situation. A convenient method which can be applied is to use accesses to non-existent memory to indicate malfunction. Both hardware and software errors can result in such accesses. Even if a fault does not immediately cause this type of access it is often very quickly followed by one eg. execution of random op-codes immediately provides potential for unspecified memory references. In practice, using accesses to non-existent memory as a means of fault detection has proved very successful.

Fortunately, provided no bus timeout occurs, the state of the UCMP system is effectively frozen immediately after attempted access to non-existent memory. Therefore, at the instant of detection, the state is recorded and can be observed by the user. Special hardware enabling observation of the state is described in section 6.3.4. If more than one interconnecting bus had been employed this would no longer be the case and more elaborate methods causing cessation of processing would be required.

6.3.2 Addressing Structure

The distribution of all MULTIBUS accessible memory within the available 1M byte address space is detailed in figure 6.5. The local memory of the master processing element is placed in the highest 32K byte

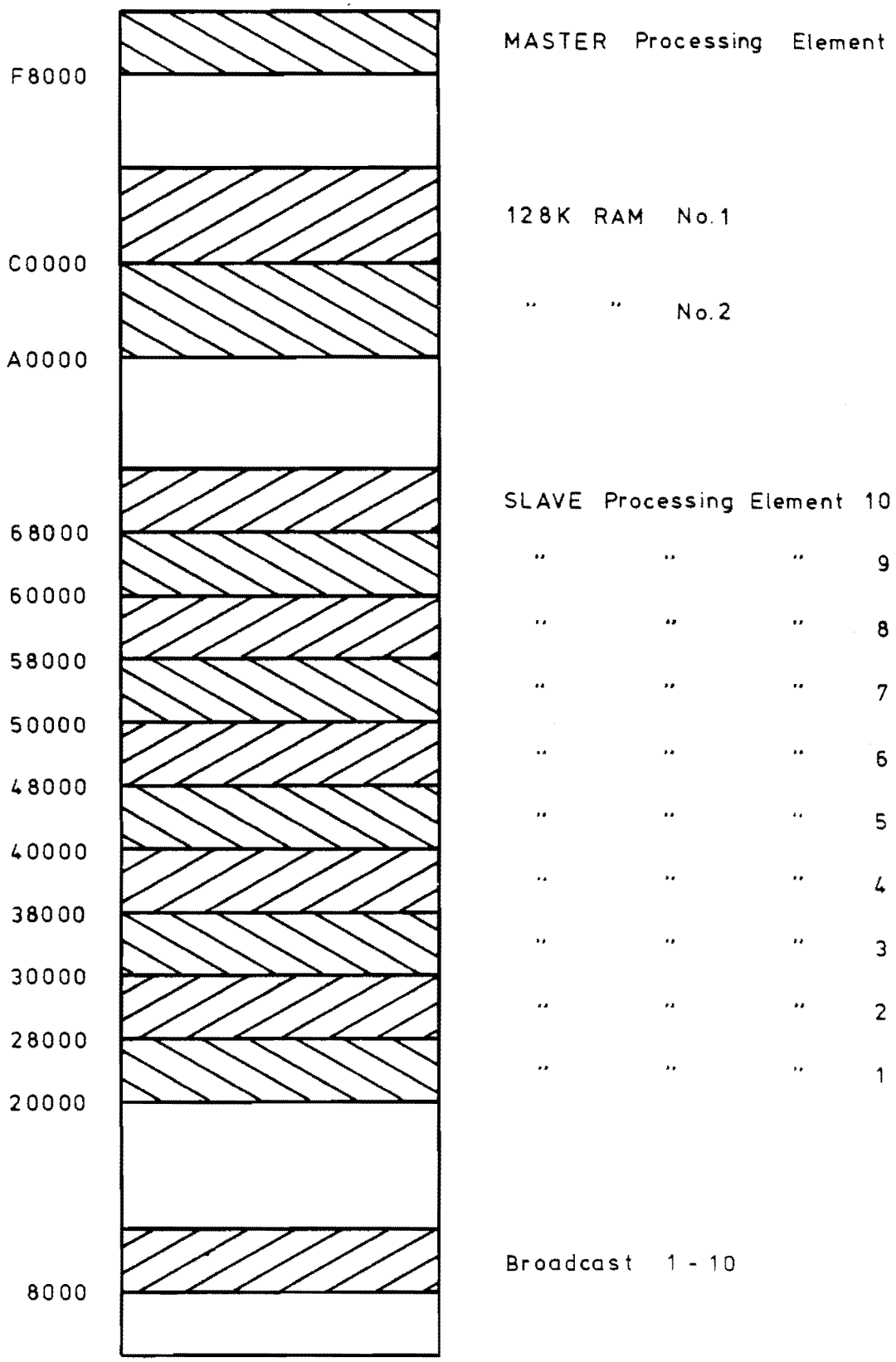


Figure 6.5: Address Space of the UCMP System as Viewed from MULTIBUS

block. It, therefore, encompasses the locations accessed by slave processors when they are reset ie. starting at 0FFFF0H. Globally accessible memory is placed below this. In practice a single 128K byte block located from 0C0000H to 0DFFFFH has been implemented with a similar area starting at 0A0000H being made available for expansion. Slave memories, of which 10 are illustrated in the figure, are distributed sequentially from a base at 020000H. The lowest 32K bytes are not used as this area is used exclusively for local references. The subsequent 32K bytes, starting at 08000H, are assigned as broadcast space. Writing to this area can be configured to result in a multiple write to all slave processing elements.

The mapping between each processor's address space and physical memory differs for all processors. Figure 6.6 summarises the differences. Each processor sees its local memory in the lowest 32K bytes which is where interrupt vectors are placed. All slave processors refer to MULTIBUS on reset and find the first executable code at 07FF0H in the local memory of the master processing element. The master processing element, on the other hand, contains local read only memory to which it refers on reset. This memory is not accessible from MULTIBUS.

As a result of this structure the following features emerge:

.all local and global memory is uniquely addressible by all processors,

.broadcast ie. all slave memories can be loaded simultaneously,

.the operation of the slave processors on reset is easily controlled by the master processor (even without globally accessible memory),

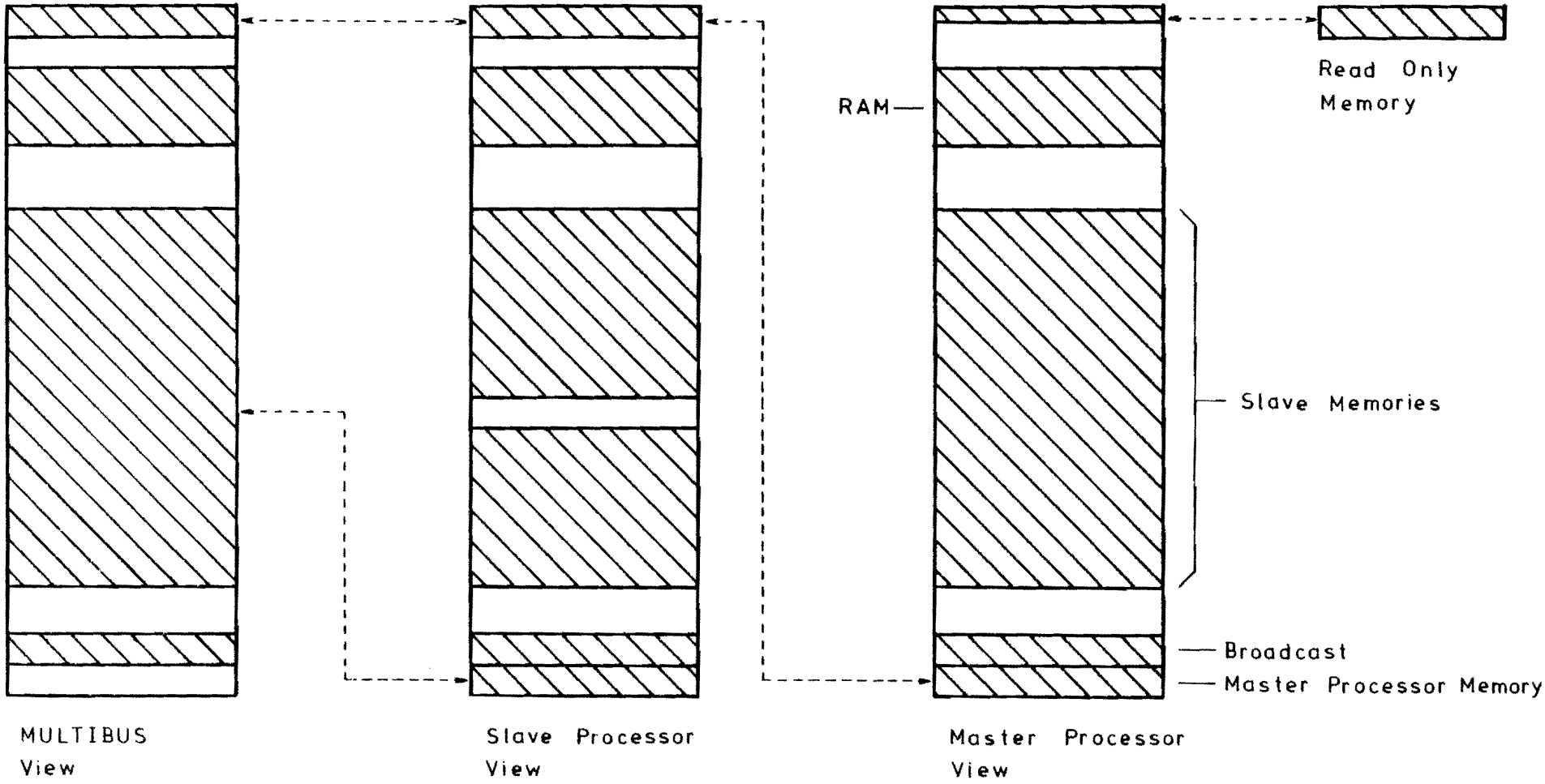


Figure 6.6: Relationships Between UCMP System Address Space Views

.interrupt vectors can all be stored locally and, as such, can differ from processor to processor,

.the operation of the master processor on reset is defined by the contents of read only memory which can contain a monitoring program which forms a minimal operating system, and

.data used by all processors can be referred to by common addresses.

6.3.3 Control Structure

The master processor supervises operation of the slaves using the control bus. As illustrated in figure 6.7, two signals enter each slave processor :- reset and non-maskable interrupt (NMI). The master processor can reset all the slaves simultaneously and interrupt them individually.

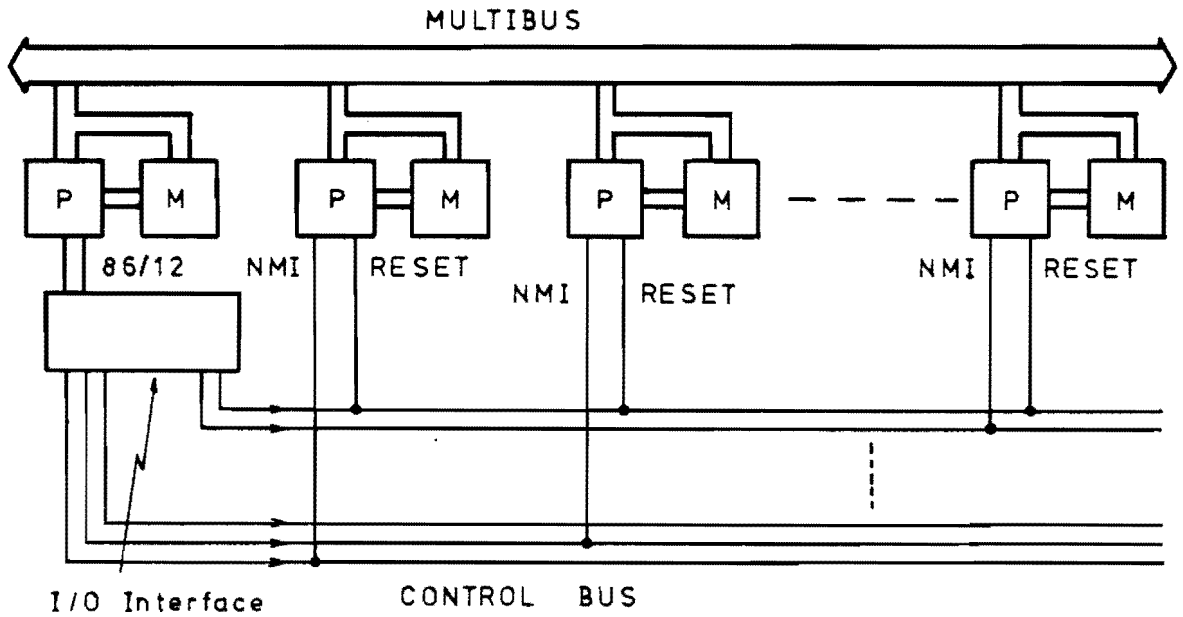


Figure 6.7: Slave Control Structure

Once reset, no slave processor can interfere with either local or globally accessible memory. This state is referred to as hibernation.

During hibernation the master processor and host processor interface have unhindered access to all MULTIBUS accessible memory. When reset is released every slave executes code found at location 07FF0H in the master processor's local memory. Typically, this will be a very short sequence, setting up stack pointers, and culminating in a halt instruction.

Slave processors can then be prompted individually using the NMI signal lines. During this interrupting process, information is written to the stack. This is why it is necessary to locate the stack on reset. Any number of processors can be initiated once they have been placed in a halt state following a reset. Operation after an NMI is defined by the contents of a vector at 08H in each local memory. At any time during execution or after a halt instruction is executed further NMI's will cause reinitiation.

From the master processor, the control bus is viewed as a series of output ports. Slave reset is level triggered and requires a single bit. NMI's are edge triggered and one bit is needed for each slave processor.

6.3.4 Basic Operation

The core components of the UCMP system can operate in a stand alone mode. In this mode no prepared code can be loaded from storage devices and very little support is available. In practice, the system is very unlikely to be used as such, but the facilities available are also common to both of the invisaged modes of operation.

A monitor program (86MON) stored in read only memory in the master processing element gives the user control via an attached terminal as was shown in figure 6.4. The master processor is initialised by the user via a MULTIBUS signal (INIT). As is detailed in section 7.4.4.1, the monitor program initialises the ports which form the control bus, and resets all slaves causing hibernation. The user can then interactively set and

observe the contents of memory and invoke the execution of code by the master processor. In turn, this could initiate slave operation.

Special hardware, shown in figure 6.8, oriented towards debugging in a multiprocessing environment, enables observation of the following:

.current address asserted on MULTIBUS

.current bus master

.enabled slave processors

A common fault, called bus jamming, serves to illustrate the value of these debugging tools. If, for any reason, a processor attempts to access a location at which there is no memory that processor will wait indefinitely for response. In this situation the master processor cannot ascertain the state of the system as it has no access to MULTIBUS. Its only option is to reset the slave processors to re-establish control, forcing the offending processor to relinquish MULTIBUS. In doing so, all information with respect to the cause of the problem is lost. However, using the special debugging hardware, the fault can be diagnosed ie. (a) the offending processor would be the current bus master, and (b) the address it was attempting to access would be asserted on MULTIBUS.

6.3.5 Operation Support Options

Two distinct modes of operation use (a) an MDS, and (b) the VAX 11/780 computer as host.

Salient operating features when using an MDS include:

.An in-circuit emulator (ICE) for the 8086 processor, ICE-86 (INTEL(b), 1982), provides facilities for the symbolic debugging of software in its hardware environment. This enhances debugging

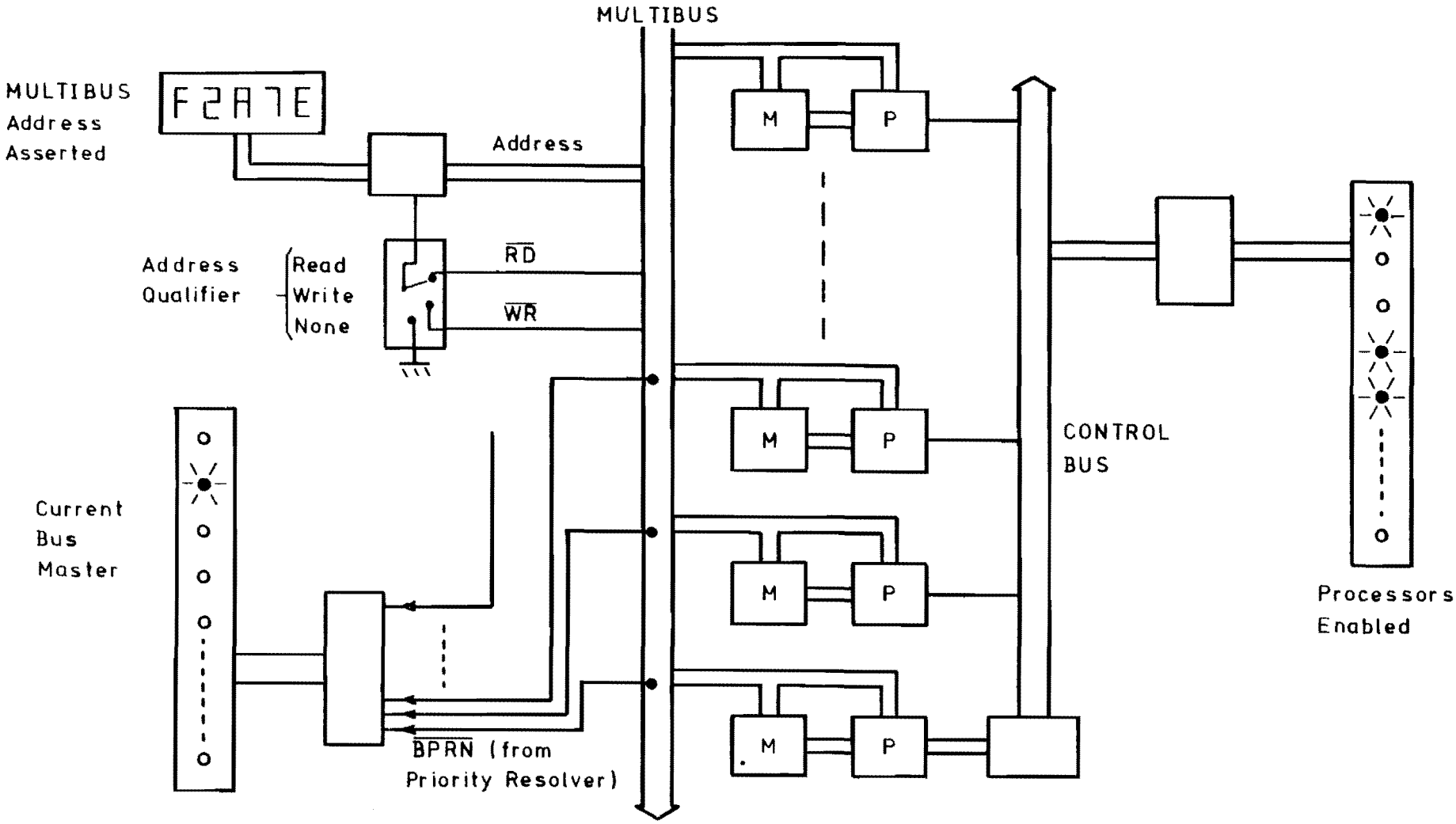


Figure 6.8: Hardware Aids for Parallel Execution Debugging

significantly when compared with conventional debugging tools which are usually programs run in the same processor as the software being tested. The omnipotent nature of ICE is of particular value when hardware can introduce errors eg. when other processors in a multiprocessor interfere in the operation of the one under test. In addition to its debugging capabilities, ICE can be used to load code stored on floppy disc.

Hence, using the MDS provides an environment suited to an iterative sequence for code preparation and debugging steps.

Alternatively, the VAX computer offers:

.a ready execution device for the UCTS program,

.mass storage, and

.very fast operation with respect to loading of code and transfer of data.

To enable transition between modes, an independent link between the MDS and the VAX computer permits transfer of all necessary information.

The manufacturers, INTEL, provide utilities which create files of located object code. Loading information within these files ensures that code is placed at addresses as required by the processor which subsequently executes that code. When this code must be loaded via MULTIBUS, however, the addresses at which it should be loaded can differ from those used by the processor executing the code. Suitable address translation can be achieved using a specially developed utility (RELOC) described in section 7.2.4.3.

6.3.5.1 MDS Based Operation

Figure 6.9 illustrates the most common configuration using an MDS. Any one of the 8086 processors is replaced by ICE-86 giving the user a view of the system as it appears from a selected processing element. The debugging facilities of ICE can then be used in cooperation with the hardware debugging tools and supervision via the master terminal.

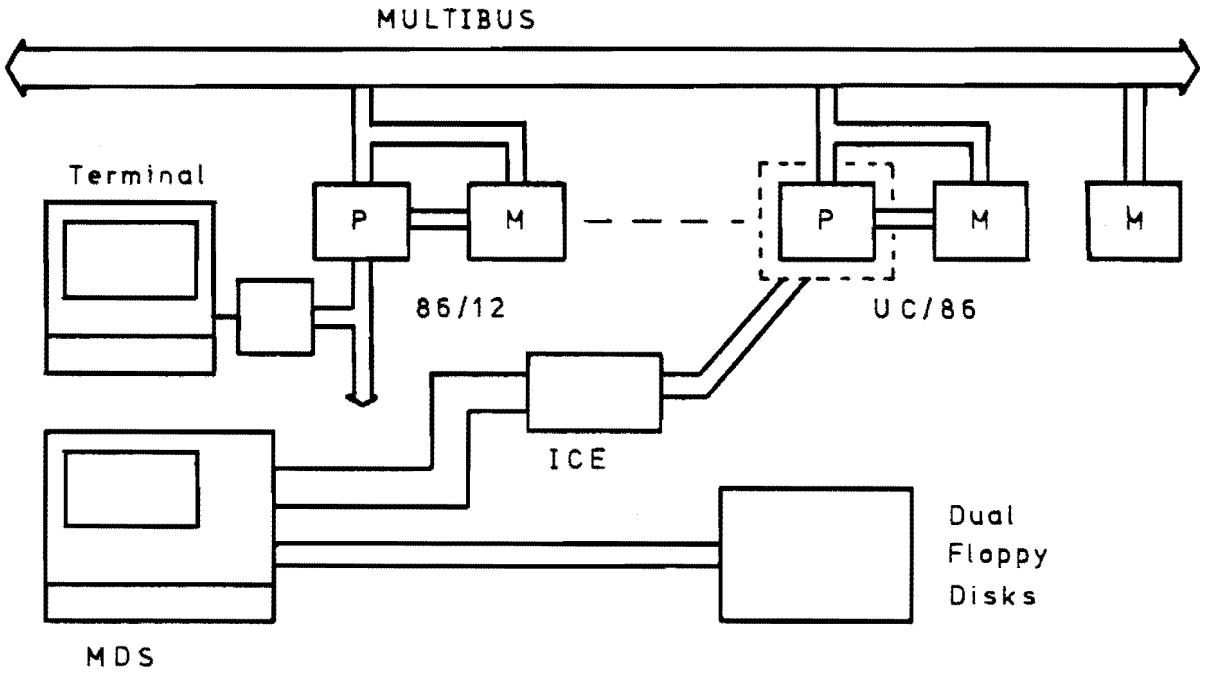


Figure 6.9: Operation with Supervision by an MDS

An alternative configuration involves substitution of the terminal at the master processor with an MDS. By executing a suitable program (86LOAD), the MDS becomes interactive and appears as a terminal. With the cooperation of facilities in 86MON, files on floppy disc can be loaded and code executed. As only limited debugging is available this configuration has not been frequently employed in practice. It does have advantages of ease of initiation, and no requirement for ICE facilities.

6.3.5.2 VAX Computer Based Operation

While using the VAX computer as host, the system is presented to the user via two terminals as shown in photograph 6.2. The function of the VAX resident terminal is defined by an interactive program executed within the VAX computer. This communicates and cooperates with 85MON, which is a small monitoring program in the host processor interface. The second terminal is that attached to the master processing element. The user is thus provided with two logically separated views of the system:

.code loading and state monitoring at the VAX computer terminal, and

.execution control and performance evaluation at the master processor terminal.

An alternative arrangement, which has been implemented as an option, has been to use only one terminal. The user can switch between the two views described above as desired. However, having only one view at an instant reduces available information with respect to the state of the system and faults are consequently more difficult to isolate. The approach has, therefore, been seldom used.

Specially developed hardware on the MULTIBUS side interfaces the UCMP system to the VAX computer. As shown in figure 6.10 commercial UNIBUS based DMA and I/O boards are employed forming the VAX computer side of the interface. The primary control and data transfer path within the VAX computer is the synchronous backplane interconnect or SBI. This 32-bit bus is interfaced to UNIBUS via a UNIBUS adapter (UBA). The host processor interface can be viewed as an inter-bus connection between MULTIBUS and UNIBUS. Two distinct, separate paths are provided.

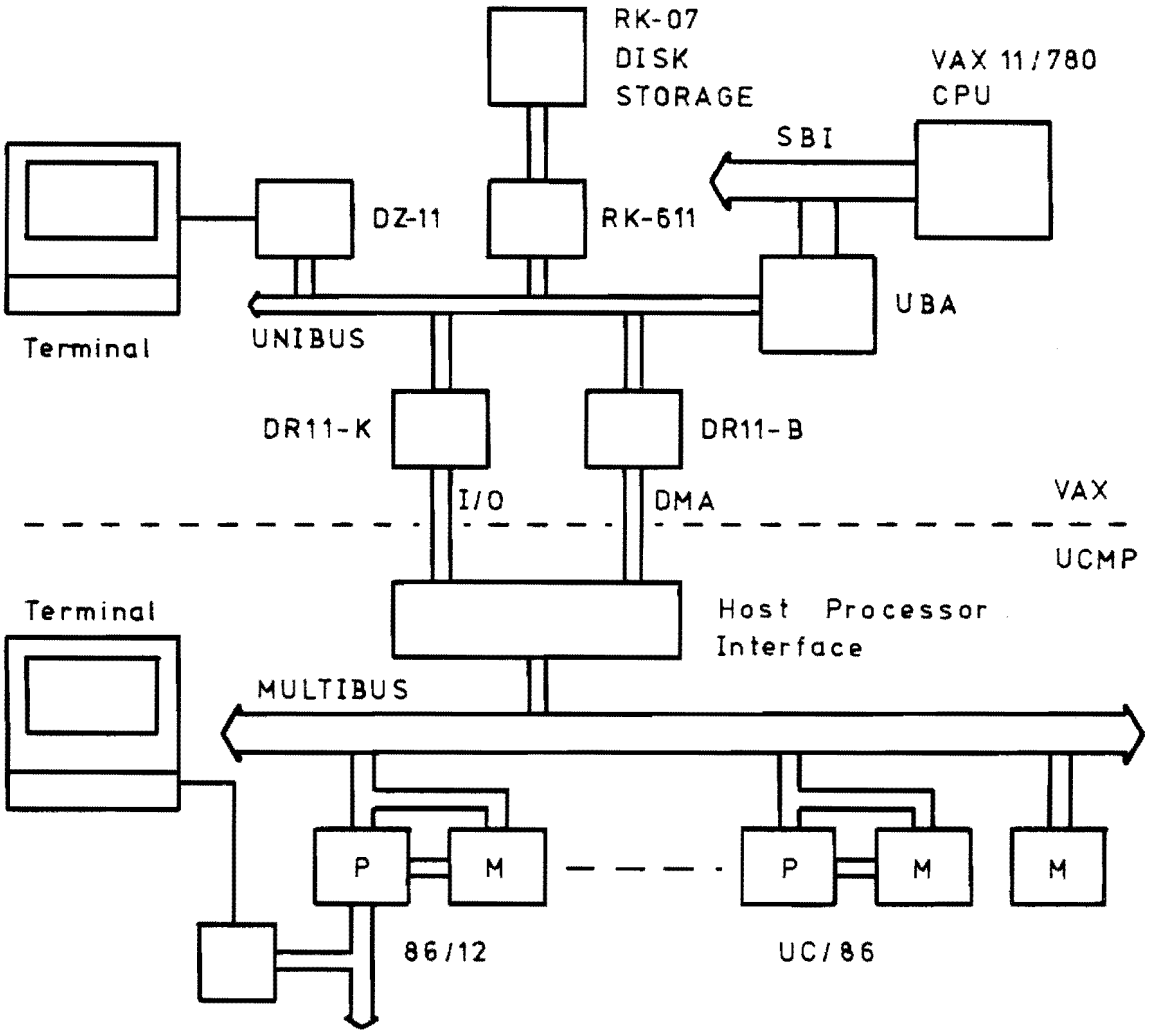


Figure 6.10: Operation with the VAX Computer as Host

Of these paths the slower I/O route involves a series of steps in transfer. Both the VAX CPU and a processor within the interface handle the data transferred. The common point between systems is seen by both as a set of ports. The alternative DMA path uses the DR11-B controller on UNIBUS and a DMA integrated circuit on the MULTIBUS side. Processors are only required to set the conditions for transfer. Subsequently, the controllers assert addresses on their respective buses and data ripples

through from one bus to the other. The I/O path offers easy control with moderate speed while the DMA route requires strict control and can reach very high transfer rates provided large blocks of code are involved. In practice, rates of around 10K bytes/sec have been typical over the I/O path. Using DMA operation, speeds approaching 1M byte/sec can be expected.

The MULTIBUS side of the interface is centered on an 8085 processor. This shares use of a local bus with an AM-9517A (AMD, 1980) DMA controller. The locally stored monitor, 85MON, uses the I/O path as a source of commands. A program run on the VAX computer , therefore, can communicate with the 8085 and, as this has access to MULTIBUS, with the rest of the UCMP system. The UCMP interactive program, INTUCMP, is described in Chapter 7. In its simplest mode this program presents the VAX computer as an apparent terminal ie. giving the user access to the facilities of 85MON. The limited tasks performed by 85MON provide the basis for expansion to an easily used, complex operating system.

6.4 Components

As was outlined in section 6.3, a number of hardware elements form the basis of the UCMP system. In this section the most important elements are discussed individually.

6.4.1 Processing Elements

Both the master and slave processing elements are based around an 8086 processor and 32K bytes of dual-ported memory. Additional needs of the slave elements are minimal while the master processing element requires many additional facilities. At the time of development, the only commercially available hardware was well suited to operation as the master processing element. As a result, an iSBC 86/12 Single Board Computer (INTEL(b), 1982) was purchased. This served not only as a master, but also

as a model to determine the exact requirements of slave processing elements. Significant savings in cost were achieved by development and implementation of slave processing elements using a new board designated the UC/86.

6.4.1.1 iSBC 86/12 Single Board Computer

Photograph 6.3 illustrates the physical layout of the 86/12. The board conforms to MULTIBUS mechanical specification. Multibus signals, including power supplies, are provided through the P1 connector ie. the larger connector on the lower edge. This forms the link to other processing elements and memory, via the backplane. Connectors on the opposite card edge provide parallel I/O forming the control bus and serial I/O to a terminal.

Bus structure is illustrated in figure 6.11. Elements used are depicted using full lines. Three separate interconnected buses are employed:

.MULTIBUS to interface to other elements,

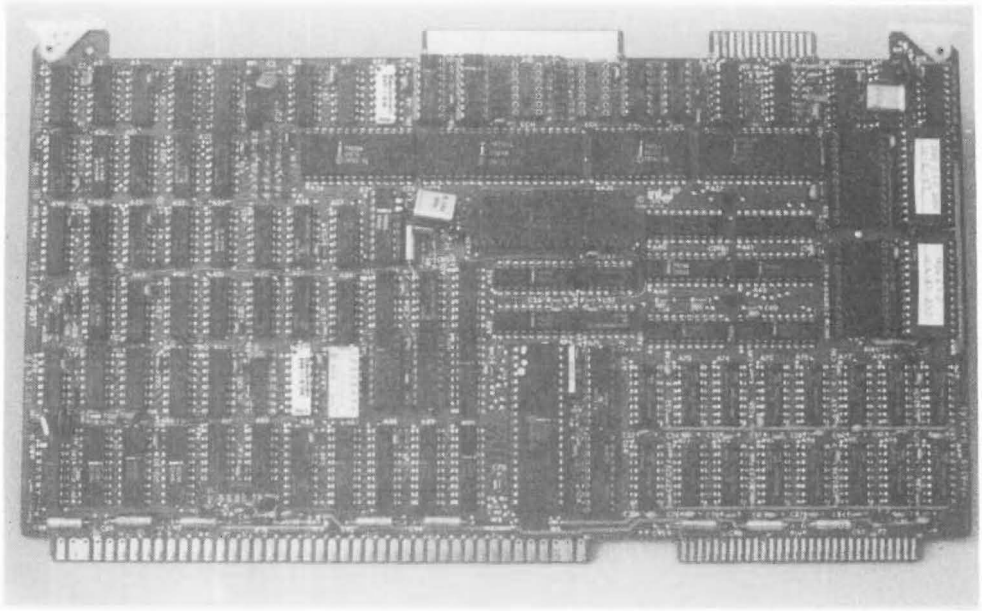
.a dual-port bus to local RAM, and

.an internal bus linking the processor to the many local peripheral and memory devices.

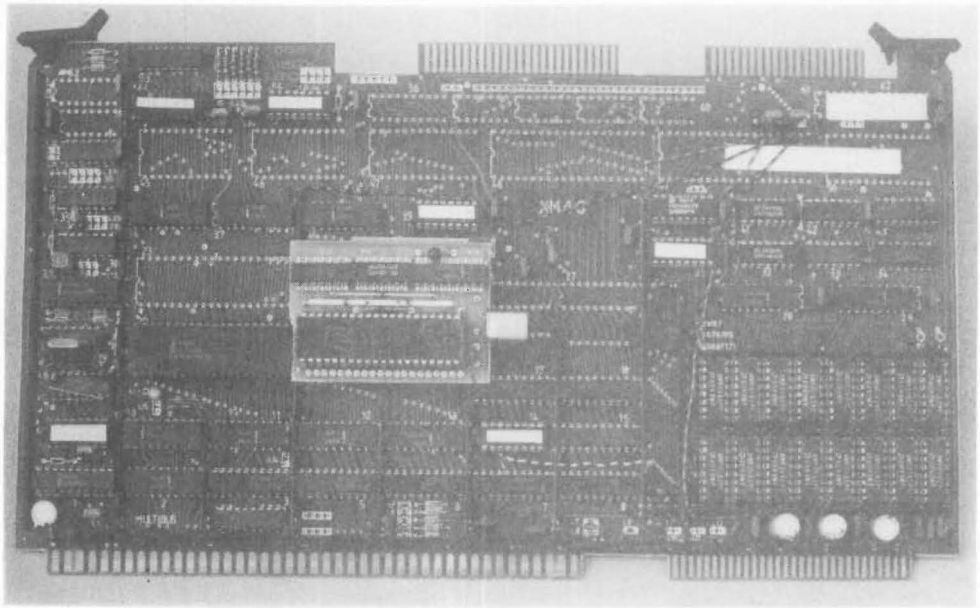
Principle components applied in the role as master processing element are as follows (Note that the abbreviations and terms used are defined at the beginning of this thesis and in Appendix 1):

.The 8086 processor

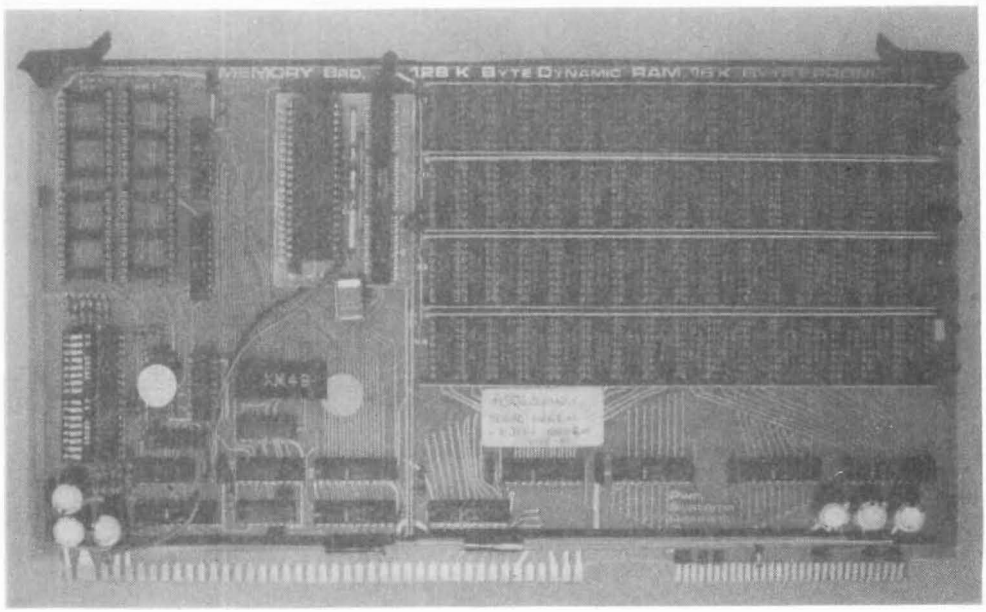
.4K bytes of the available 16K byte area of read only memory are used to store 86MON (see Chapter 7). Two 2716 EPROM's



Photograph 6.3: iSBC 86/12 Single Board Computer



Photograph 6.4: UC/86 Single Board Computer



Photograph 6.5: Memory Board

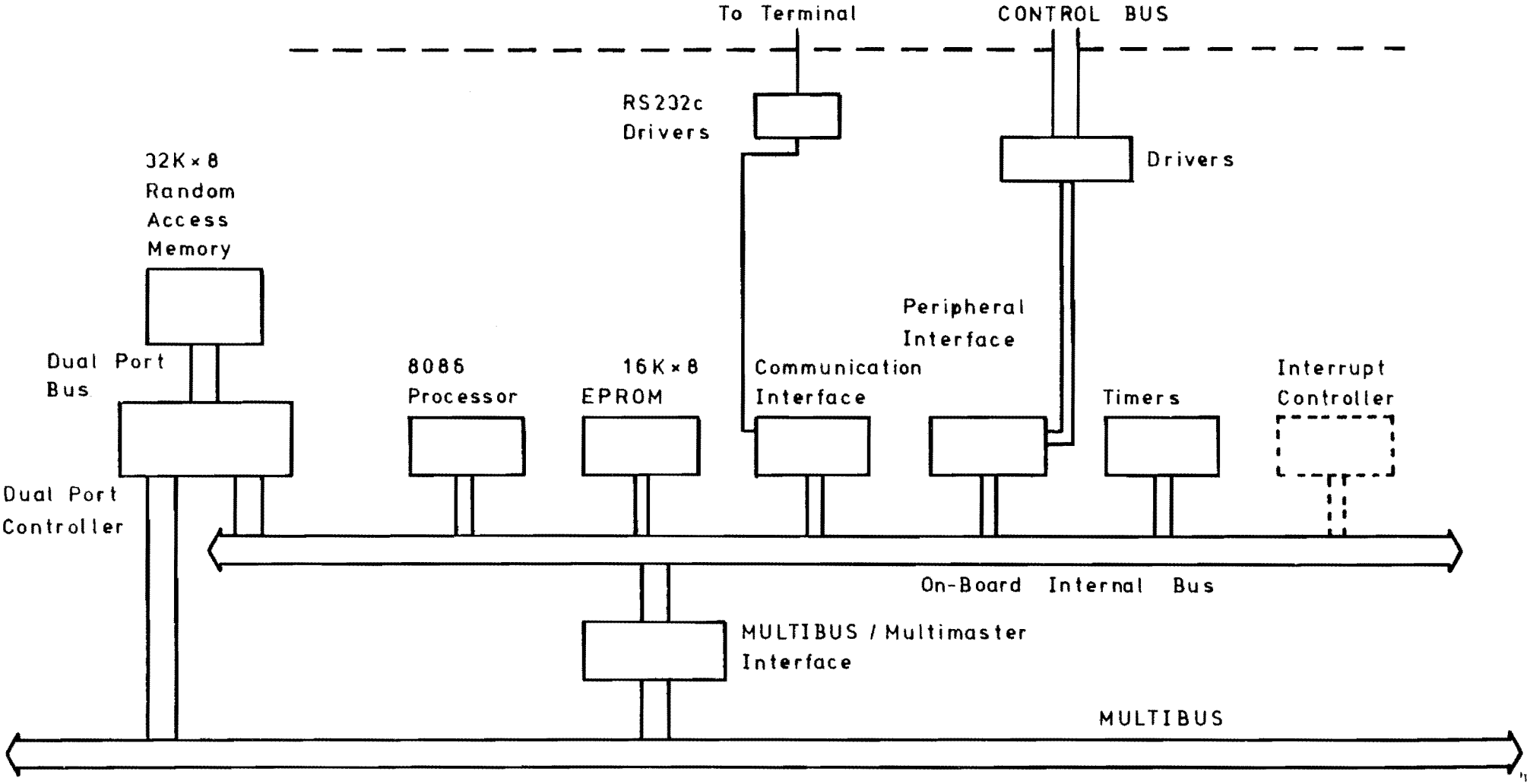


Figure 6.11: Structure of the iSBC 86/12 Single Board Computer

(INTEL(a), 1982) are used.

.An 8251A Programmable Communication Interface (PCI) forms the basis for serial communication (RS232C compatible). Baud rates and other characteristics are programmable.

.Up to 24 independent signal lines are provided by parallel I/O forming the control bus. An 8255A Programmable Peripheral Interface (PPI) is the principal element used, with the lines addressed as three ports.

.Timers which can operate independently with respect to the processor are provided by an 8253 Programmable Interval Timer (PIT). These are employed in performance evaluation

.The dual-port controller and 32K bytes of RAM provide code and local data storage. An 8202 dynamic RAM controller and sixteen 16Kx1 dynamic RAM IC's are employed.

.MULTIBUS interfacing, based on the 8288 bus controller and 8289 bus arbiter, permits multiple master operation.

6.4.1.2 UC/86 Single Board Computer

Physical layout of the UC/86 board is illustrated in photograph 6.4. Mechanically it is similar to the 86/12 with respect to position of card edge connections. A different approach to construction is apparent, however, in that (a) only two wiring layers are used, and (b) power supplies are distributed using special decoupled conductors soldered to the card. Much use is made of programmed logic arrays to simplify the board by reducing the density of packages.

As shown in figure 6.12, the bus structure is similar to that used

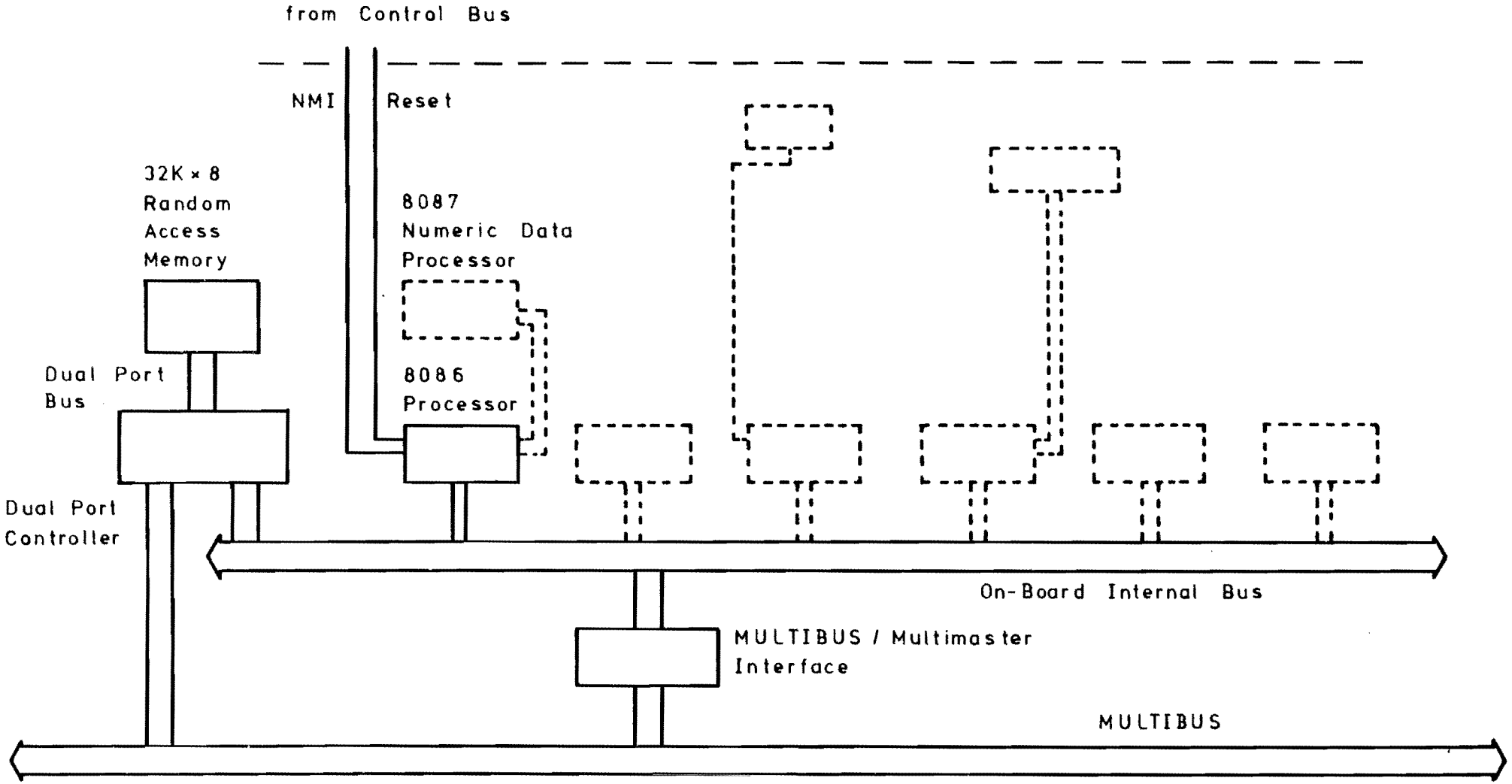


Figure 6.12: Structure of the UC/86 Single Board Computer

in the 86/12. An important additional feature is the provision of space for inclusion of an 8087 numeric data processor. If all components are installed, the board has features similar to those of the 86/12. However, in its role as a slave processing element far fewer packages are necessary. The saving in components is illustrated by comparison of photographs 6.3 and 6.4.

Control bus signals enter via the P2 connector, which is adjacent to the P1 connector. Optional jumpers can interrupt the signals in their path to the processor. These allow a choice between local operation, as would be used with ICE-86, and control by the master.

6.4.2 Memory

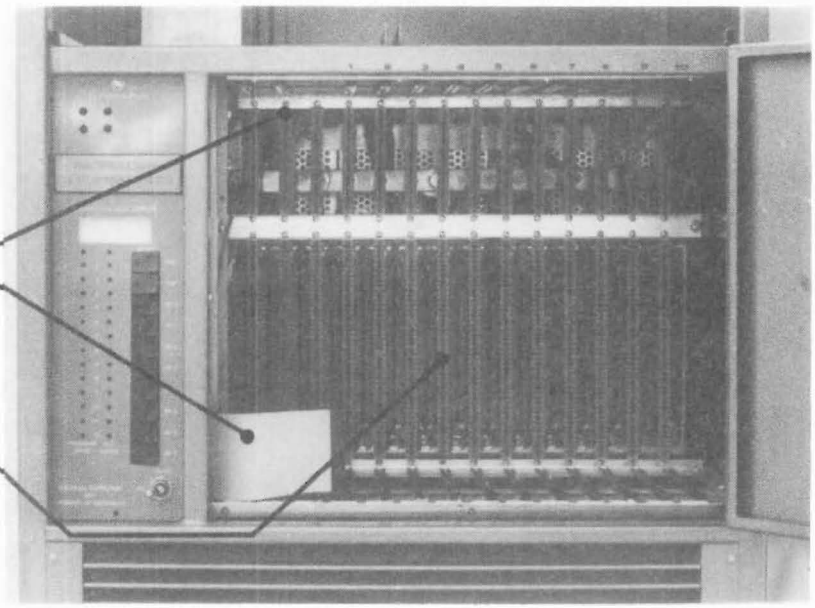
Globally accessible memory is implemented using a 128K byte MULTIBUS compatible board (Barth, 1981c). Photograph 6.5 shows the layout with four banks each with sixteen memory IC's. Decoding on the board allows placement of the 128K bytes in any one of eight continuous blocks. Many similar boards are now commercially available.

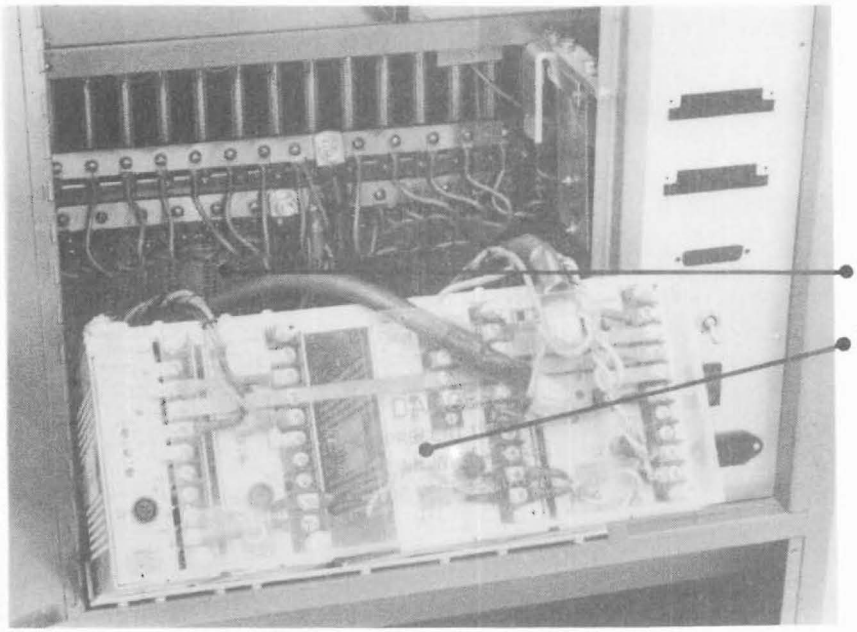
The board uses the same controller and dynamic RAM IC's as the processing elements. The 16Kx1 memory IC's are rapidly being overtaken in cost effectiveness by 64Kx1 RAMs. Consequently, commercially produced boards, which provide half a megabyte of memory using a similar area of printed wiring assembly, are now available.

6.4.3 Backplane

The set of conductors required to connect the P1 edges of all MULTIBUS boards is realized on a printed circuit card called the backplane (Parr, 1980). The physical form of the card, as it is situated within the UCMP system, is illustrated in photographs 6.6 and 6.7. Bus arbitration is

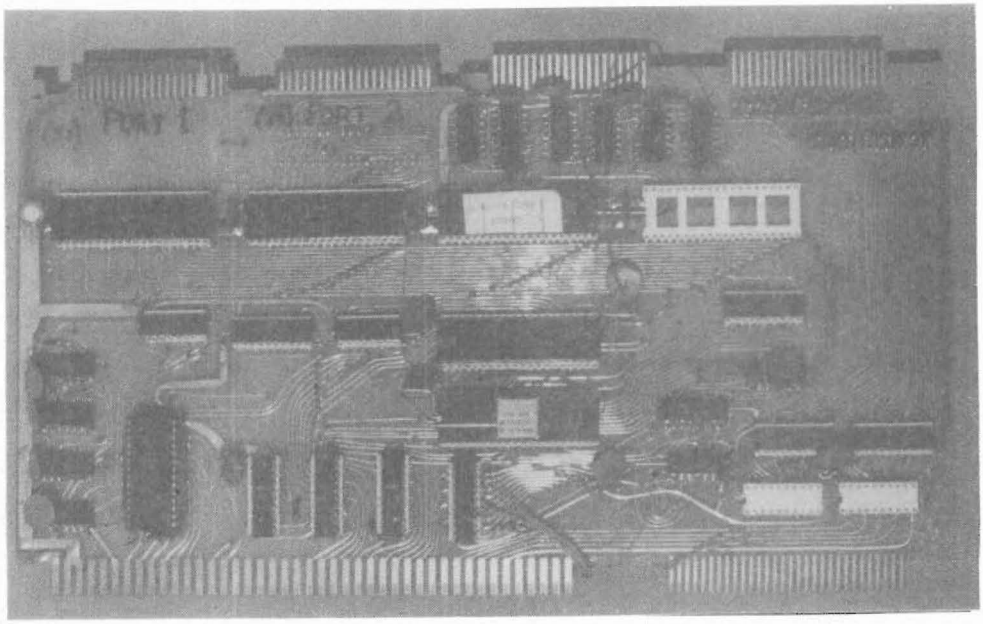
Photograph 6.6:
Backplane and
Control Bus





Photograph 6.7:
Backplane and
Power Supplies

Photograph
6.8:
Host
Computer
Interface
Board



implemented using circuitry at one end while bus terminating resisters are situated at the other.

Priority resolution occurs at the start of each MULTIBUS cycle. Time taken in arbitration, therefore, adds to access times and is best minimised. Hardware implementing arbitration must reach a stable state after a change in requests within the period of MULTIBUS's synchronising clock (BCLK). The serial priority resolution scheme implemented allows operation at the maximum permissable clock frequency (ie. 10MHz).

Power is provided to all boards via the backplane. To reduce noise, arising because of the resistance of power supply cabling, very strong conductors close to the card are connected individually to each slot. With a current requirement of approximately 50A at 5V power supply noise reached unacceptable levels without this arrangement.

6.4.4 VAX Computer Interface

This section outlines a stand alone single board computer specially developed to cooperate with the DR11-K and the DR11-B in the transfer of data between MULTIBUS and UNIBUS. Minimal interference to the operation of core system elements is achieved through the use of a local processor and memory. Not only does this approach remove any need for external processing, but it also reduces accesses via common resources to those which are essential. The board, which is itself a MULTIBUS master, implements the I/O and DMA data paths introduced in section 6.3.5.2.. It is illustrated in photograph 6.8. A detailed description of the board was written by D. Low (1980).

An 8085 processor (INTEL(e), 1978) cooperates with an AM9517A DMA controller (AMD, 1980) to implement these paths. Figure 6.13 illustrates the bus structure, I/O, and memory employed. The local minimal operating

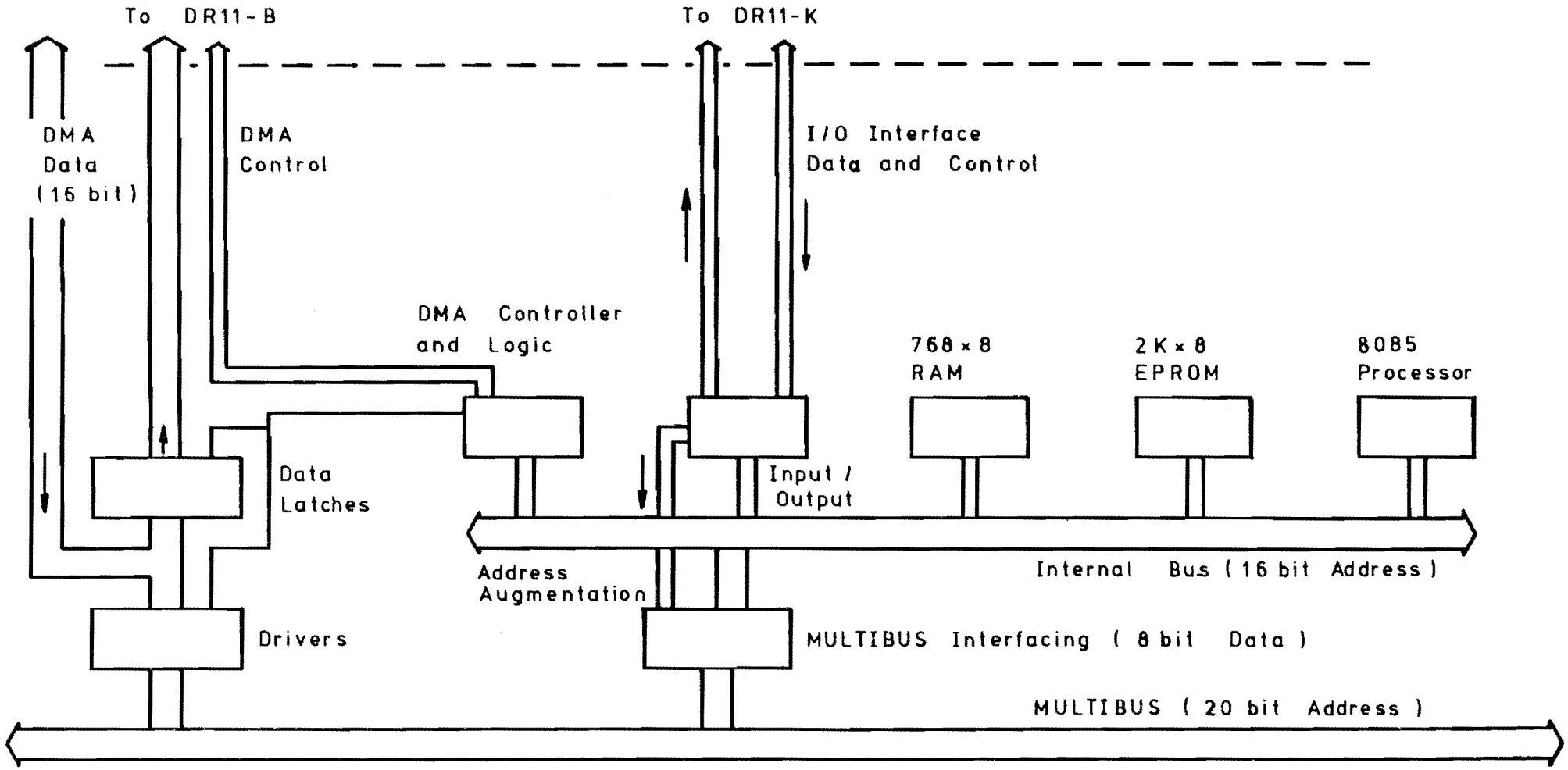


Figure 6.13: MULTIBUS-VAX Interfacing Single Board Computer with DMA Controller

system, 85MON, is stored in an 8755 EPROM, which is also used to provide address augmentation via I/O ports. Local RAM is provided by three 8155's. I/O also on these is used to interface to the DR11-K. The MULTIBUS interface is based on an 8218 bus controller which is compatible with the 8080 microprocessor. Although a similar device, the 8219, is more suited to use with the 8085, the nature of control signals on the AM9517A ensures that the 8218 is, on balance, preferable.

The limited direct addressing of the 8085 creates the need for address extension during MULTIBUS accesses. The 64K byte address space, as seen by the 8085, is shown in figure 6.14. The lower half is reserved for local reference leaving 32K bytes as a current page in MULTIBUS memory. Five bits set through an output port define the page within MULTIBUS's full address range seen by the 8085. Minimal, but adequate, address decoding leads to the non-consecutive spread of local RAM throughout the low 32K bytes.

Hardware implementing the DMA data path has been built but, to date, has not been used. Reasons include firstly, that the speed of the I/O path has proved more than adequate in the tests made, and secondly that, at the time of development, the DMA controlling hardware within the host was not available.

6.4.4.1 Input/Output

Signals to and from I/O pins on two 8155's interface to corresponding signals provided and expected on the DR11-K (DEC(c), 1975). Additional current drive is provided for signals sent to the DR11-K. Separate cables (see photograph 6.1) form 16 bit data paths in each direction.

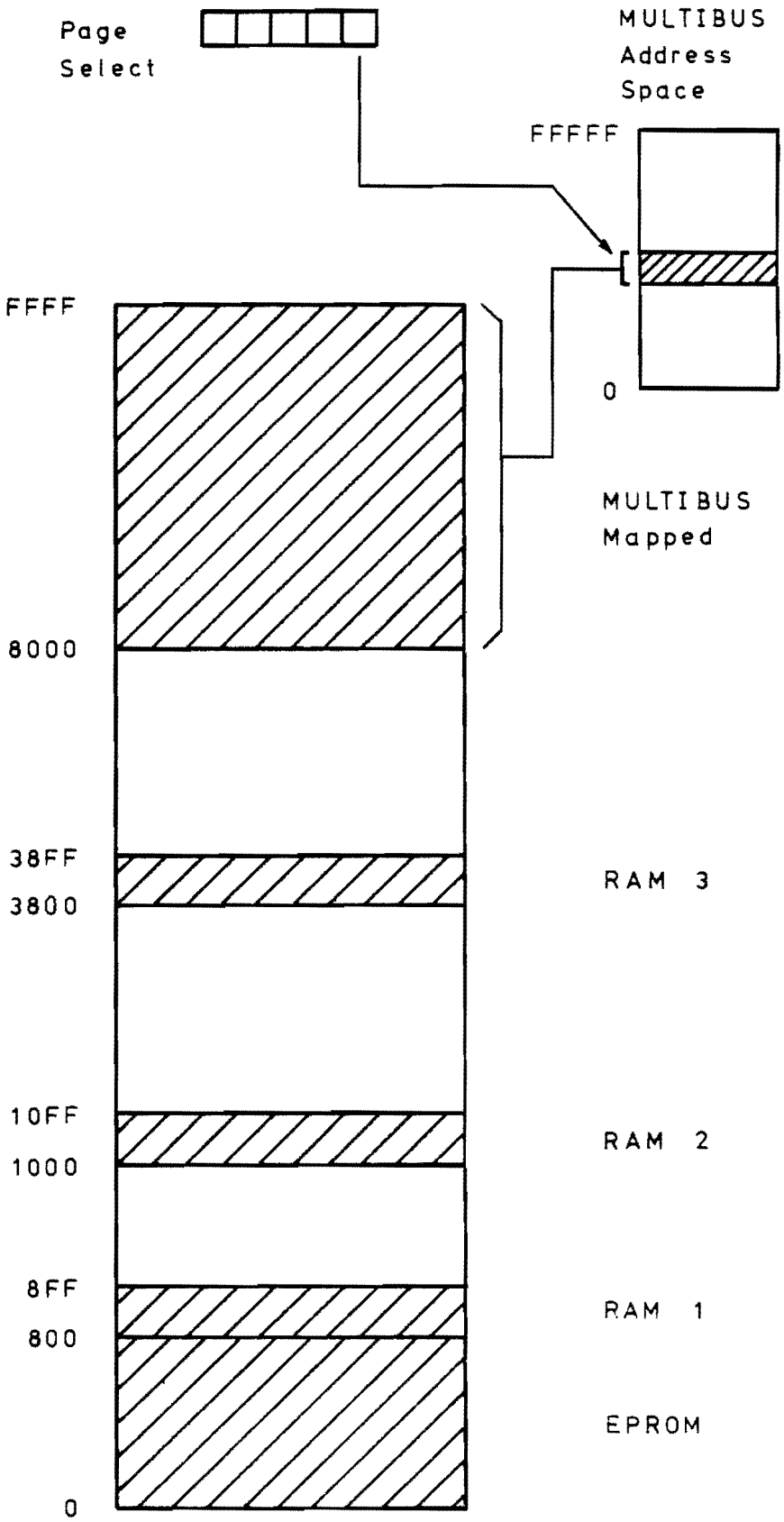


Figure 6.14: 8085 Processor's Address Space

Both the VAX computer's central processor and the 8085 see their respective data paths as I/O devices. Protocol is based on polling of status bits set and reset as data is transferred. As the 8085 need not pass data received on to MULTIBUS, the link can be used to pass instructions from the VAX computer to the 8085 without interference to the core UCMP elements. Similarly, response can be given without disruption. When data is transferred to and from MULTIBUS the 8085 can only do so in single byte quantities. As there is considerable code executed implementing the protocol of transfers anyway, speed reduction is insignificant when compared with 16 bit transfers.

6.4.4.2 DMA

Once initialised, the AM9517A DMA controller coordinates the MULTIBUS side of DMA transfers. Various control signals ensure cooperation with the DR11-B while each 16 bit word is passed through a sequence of buffers from UNIBUS to MULTIBUS and vice versa. Source and destination addresses are provided by the respective controllers.

Initialisation of the DMA controller is done by the 8085 which addresses it as a series of I/O ports. As both the 8085 and the AM9517A are potential masters of the local bus, arbitration for use of the bus is necessary. Protocol allowing exchanges is implemented using the Hold and Hold Achnowledge facilities of the 8085.

Like the 8085, the AM9517A can only address 32K byte pages on MULTIBUS. Because the 8085 must be involved in page changes this restricts block size for DMA transfers to 32K bytes. As this corresponds to the local memory size of each core processing element, it is not an unreasonable limitation.

6.5 Summary

A multiprocessing system, developed to execute and test algorithms related to transient stability analysis, has been described. Its homogeneous, flexible structure and extensive memory make it applicable to wide range of problems.

The system is based around INTEL 8086 microprocessors. Many features, including cost-effectiveness, compatability with transient stability related data quantities, and multiprocessing, ensure the suitability of this processor. Both the modest speed, and the limited ability to handle extended word-length floating point numbers, of the 8086 can be enhanced using 8087 numeric data coprocessors.

Considerable effort has been directed at enabling effective program production and testing. Existing computer hardware has been used where possible to host the system and provide a program development environment. Commercially provided debugging aids are augmented by a number of hardware features facilitating the isolation of problems during parallel execution.

CHAPTER 7

UCMP SYSTEM SOFTWARE UTILITIES AND OPERATION

7.1 Introduction

In serial computer applications, compilers provide a means of increasing both programmer productivity and code integrity. Slight, often insignificant, overheads are introduced by way of increased execution times and greater storage requirements ie. when compared with optimal code selection achievable using low level languages. Run time execution sequence is very closely related to the program flow at source level.

Three methods are identified to translate operational requirements to machine code in a parallel processing environment:

.compilers which determine processing distribution,

.user directed compilers, and

.serial compilers.

When the compiler determines distribution of processing and establishes points of synchronisation etc., source code similar to that prepared in serial applications can be translated into parallel executable code. A description of various compilers in this category (Lau et al, 1982) concludes that performance expectations are low. Reasons include the problem that optimal serial programs do not necessarily approach situations in a way that is optimal in parallel. Also, there may be many levels at which parallelism exists and can be exploited, but the best choice of level, which may be crucial, is very difficult to make without experience.

A number of languages, eg. Ada, PL1, and Concurrent Pascal, include constructs that explicitly direct which processes can occur simultaneously. As such, the programmer defines the distribution of processing and the order of execution while writing a single source program. Concurrent Pascal, for example, is used in the commercial DEMOS system where an important design requirement is that parallel programming skills must be acquired quickly. After an iterative development cycle, performance improvement is achievable at the expense of increased programming difficulty.

Where a translator with capability to produce multiple programs suited to parallel processing is not available, similar code can be developed using serial compilers. Separate programs need to be prepared for each processor, and communication and synchronisation arranged through explicit source statements. Secure interprocessor communication must be facilitated, for instance, through definition of where semaphore operations should occur.

To date, serial compilers have been employed in operation of the UCMP system. Although these represent the most involved approach to program implementation, their use has not presented a great disadvantage. Programs implemented have been relatively short, and an in depth knowledge of execution sequence, as implied by the method, is of value. The manufacturers of the processors employed in the UCMP system support Pascal and Ada. Therefore, parallel program compilers may become available enhancing the existing range of development possibilities.

This chapter describes the environment in which software for the UCMP system can be prepared, debugged, and tested. To allow efficient program preparation, and permit fast code testing under a variety of conditions, a number of distinct hardware configurations are supported.

7.2 Programming Environment

For a given problem, the programmer must decide what distribution of processing is to be implemented, and then prepare separate programs for each processor involved. Consideration must be given to methods of coordination of processors and maintenance of data integrity eg. through the use of semaphores. No automated support is provided to implement these functions, but available facilities simplify their inclusion.

The environment under which these programs run is depicted in figure 7.1. Each program makes use of local memory for stack and intermediate result storage. Synchronisation with other processors and access to global data is achieved through links to globally accessible memory. Although it is possible, access to the local storage areas of other processors is not illustrated.

7.2.1 Address Structure

All processors view their address spaces as illustrated in figure 7.2. Areas at which memory exists but which are not accessed by user-written programs are the local memories of all other processors and the area encompassing the reset location ie. 0FFFF0H. Executable code, stack, and local data are located in the lowest 32K bytes. Shared data and synchronisation information is stored in the 128K bytes above 0C0000H.

All code, data, and stack contents are prepared as blocks, called segments, each of which is contiguously located at run time. Because there are two distinct classes of data, ie. local and global, at least two segments of data are necessary. All processors view globally accessible memory at the same locations. As a result, a practical and reliable method of definition of the contents of globally accessible memory uses an absolute definition of the location of values stored at program source

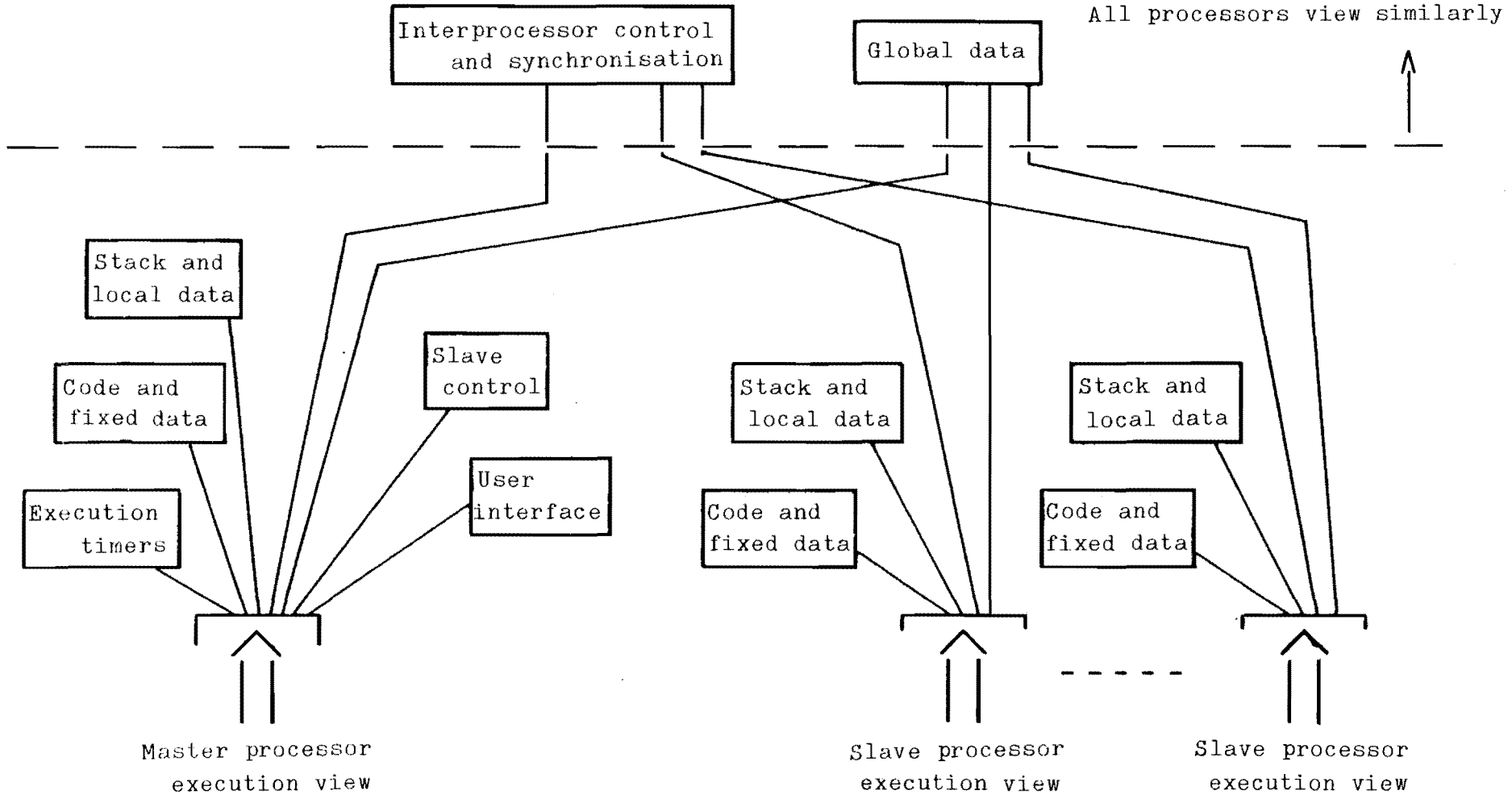


Figure 7.1: Slave and Master Processor's Program Execution Environments

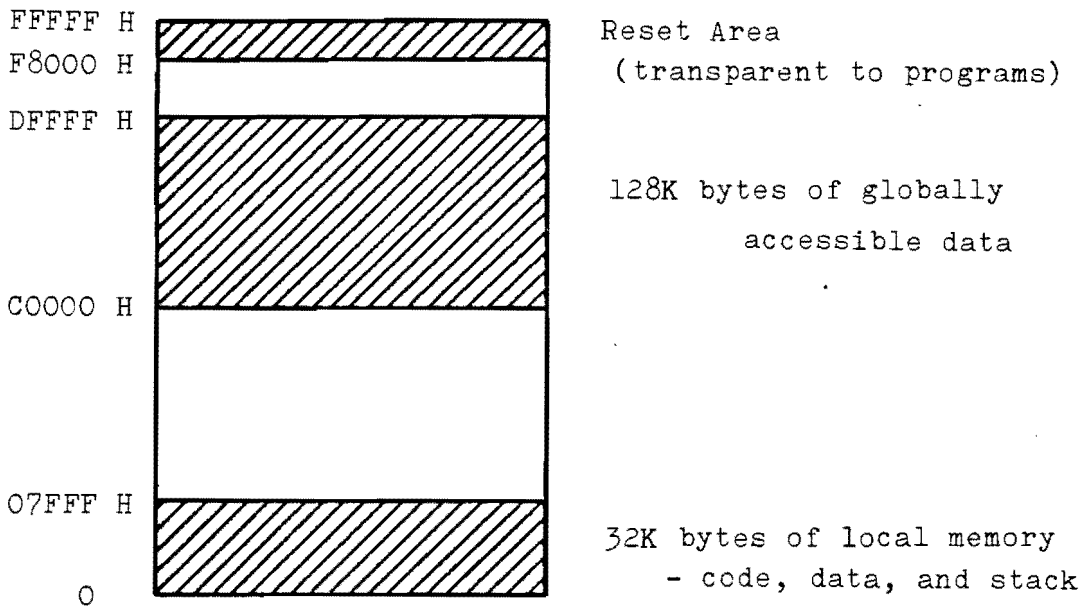


Figure 7.2: Address Space Available to Each Core Processor

level. This set of definitions can be included in each processor's program. Linking and locating modules in different ways cannot then affect run time location.

Apart from definition in source programs, data can be provided by the host processing system after execution of the UCTS program combined with appropriate data extraction routines. To transfer this data to the UCMP system, its load addresses must coincide with the absolute locations specified in each processor's source program.

7.2.2 Hierachy

Homogeneity in the bus structure implies that there is no functional difference between processors during execution and, therefore, that there is no binding requirement for differences of function and corresponding different programs. The programmer can apply any desired priority to selected processors in the chosen approach to a problem.

In addition to any programmer-defined hierachy, there is a hardware specified relationship between master and slave processors which implies differences must exist between their respective programs. It does not, however, affect any run time hierachy as all of the necessary differences can be executed before and after periods of parallel processing. The master processor's program must transfer signals via the control bus to initiate operation of selected slaves. In addition, interfacing to a user permits monitoring of performance and selection of, for instance, a revised number of processors.

In practice, there are a small number of strictly serial steps which must be taken during execution. By including these in the master processor's program, all of the slave processors can be assigned identical code. It is only a matter of convenience, therefore, that the controlling master also executes serial sections of code.

7.3 Code Preparation, Debugging, and Execution

An efficient code development, testing, and execution environment is provided by a combinition of:

.hardware execution, storage, and debugging devices,

.communication links, and

.software utilities.

7.3.1 Data Flow

Figure 7.3 summarises the software utilities and illustrates the paths between them. Data flow direction is depicted by arrows and, where appropriate, indication of the format (see section 7.4.1) in which data exists is given.

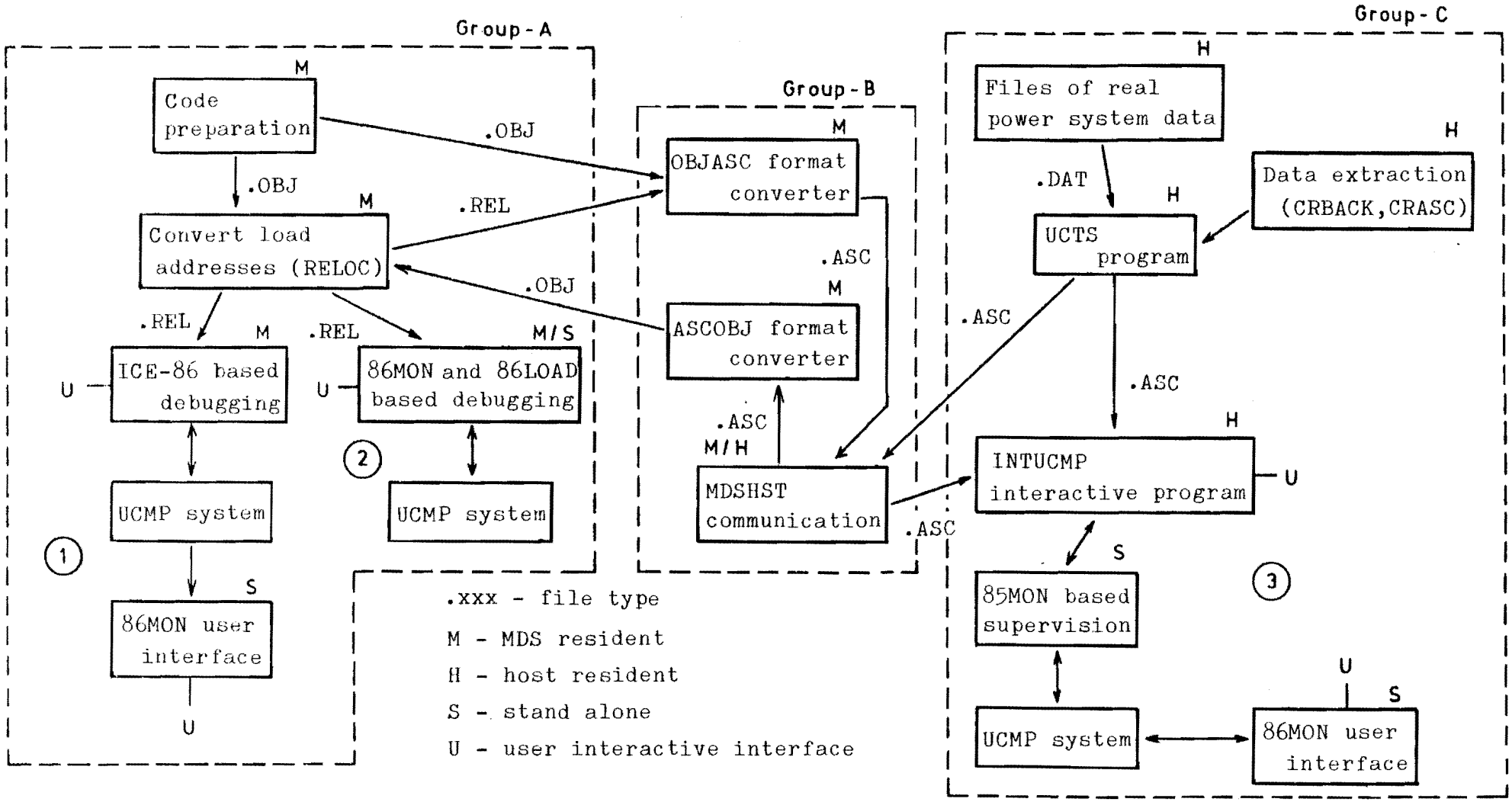


Figure 7.3: Utilities Available for Code and Data Preparation, and Program Execution and Debugging

The three distinct utility execution environments are:

.execution under the ISIS-II operating system on an MDS (such utilities are labelled 'M' in figure 7.3),

.execution under VAX/VMS on the VAX host (labelled 'H'), and

.stand alone operation (labelled 'S').

Functionally, the utilities can again be divided into three groups:

.'A'-MDS based code development and debugging. Code can be executed in either of two configurations, marked '1' and '2'.

.'B'-a set of utilities facilitating data transfer between the host and MDS systems, and to make suitable format conversions as required for these transfers.

.'C'-host processor based testing and data preparation. Code is executed here in configuration '3'.

The following section serves to both describe the available utilities and outline the sequence in which they would typically be applied.

7.3.2 Development Cycle

Serial program development involves an iterative sequence of code preparation and debugging. Parallel programs offer considerably increased scope for introduction of mistakes. These frequently occur at points where processors should synchronise and communicate. As the programs developed can run on a single processor, two levels of debugging, shown in figure 7.4, emerge. Firstly, serial debugging, as for normal program development, is required. Outside this, a second iterative loop involving

parallel debugging is needed.

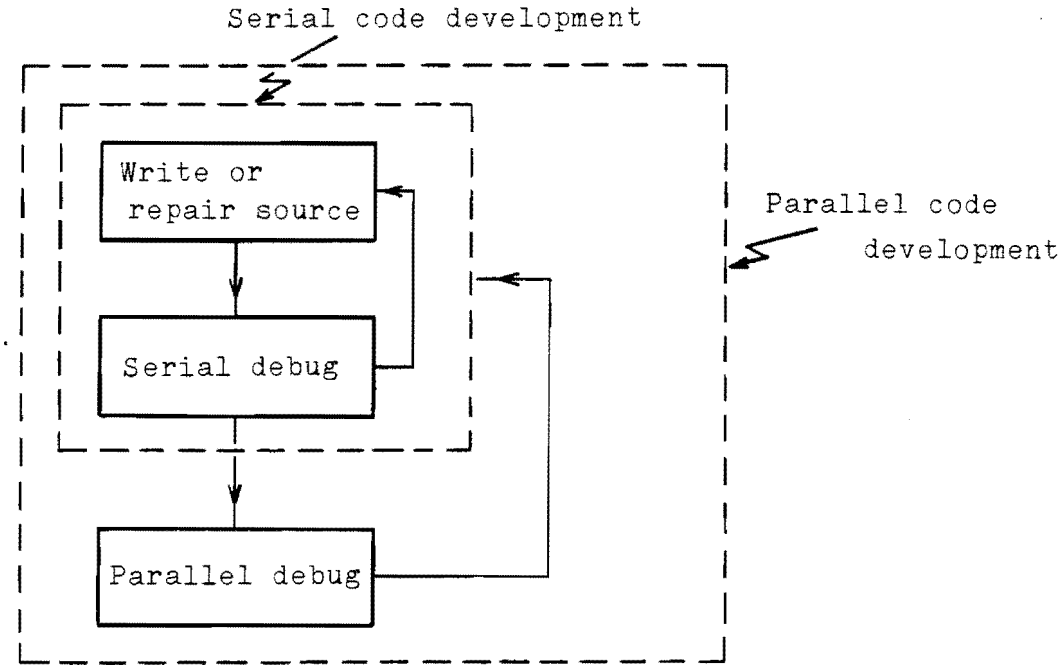


Figure 7.4: Parallel Program Development Cycle

7.3.2.1 Code Preparation

Compilers and an assembler, run under ISIS-II, allow creation of relocatable object code. This can be combined with similar code generated from other source programs, or supplied libraries, by a linker producing code which is still relocatable. Absolute location of this code is then provided by a locater.

The supplied locater produces object files with coincident load and execution addresses. If an alternate form is required, such as is needed when loading memory in processing elements via MULTIBUS, a utility called RELOC can be applied.

7.3.2.2 Serial Debugging

Suitably located code, generated with a symbol table, can be loaded and debugged in a single processing element. The in-circuit emulator (ICE) offers a wide range of software (and hardware) debugging capabilities. Features include real time execution, single stepping, breakpoints, and a large trace facility. Users may utilise symbolic referencing of memory by variable names, labels, or source file line numbers.

Alternatively, the less sophisticated features of 86MON, a monitoring program residing within the 86/12 single board computer, can be applied. Apart from its more limited facilities, 86MON lies within the processor's address space and utilises interrupts. Consequently, this debugging tool can itself introduce errors.

7.3.2.3 Parallel Debugging

Ideally a user involved in debugging should be presented with the state of the system at any instant, and should be able to adjust the state as desired. During simultaneous execution of a number of related tasks, the number of possible states is very high and particular states are often difficult to re-enter if information is lost. Available debugging tools provide a number of views of the system, but fall short of a complete state description. As a result great care in source preparation is valuable especially to ensure the integrity of interprocessor operations.

Debugging tools include:

.ICE-86,

.86MON,

.current MULTIBUS address display,

.current MULTIBUS master identification,

.current active slave processors display, and

.special program features, executed during multiprocessing operations, recording execution sequences.

All of these may be applied simultaneously. ICE-86 and 86MON have already been mentioned and the special hardware tools were described in Chapter 6. Detailed checking of execution sequence correctness is only possible using special program sections which can record the progress of every processor. Unfortunately, this method imposes a time consuming overhead which possibly affects operation and thus distorts performance measurements. In practice, however, only a very small proportion of processing time is required to make adequate recordings.

7.3.2.4 Execution and Performance Evaluation

Having become satisfied with both serial and parallel debugging stages, measurements of execution efficiency can be made. Processor numbers can be varied and data sets changed. The most appropriate environment for large numbers of measurements is under supervision by the VAX computer, making use of the INTUCMP and 85MON utilities. Parallel execution is initiated and monitored through 86MON in cooperation with the control and interactive support routines executed by the master processor. Timing enabling performance evaluation is also provided by master processor routines which utilise hardware counters on the 86/12 single board computer.

Using the same features executed by the master processor, but under control of an MDS, the same measurements can be made using either ICE-86 or a combinition of 86MON and 86LOAD. Disadvantages in both configurations

arise as a result of the slow rate at which information is loaded.

7.3.2.5 Data Changes

Without code changes, a variety of sets of data can be involved in execution. Two sources of data are possible. The simpler method of inclusion is definition at source level in code development. This is valuable where very few changes are invisaged, especially during debugging. Alternatively, data can be prepared on the host processor. Sections of the UCTS program are executed, and the CRBACK and CRASC utilities are used to produce ASC formatted files of data suitable for loading to the UCMP system.

ASC files of data can be loaded directly to the UCMP system if configuration '3' (figure 7.3) is used or can firstly be transferred to an MDS. Combined utilities, forming a serial tranfer mechanism called MDSHST, allow movement of files to an MDS. ASCOBJ translates the contents to a form suitable for loading in either of configurations '1' or '2'.

7.4 Utility Software

All program code required for operation of the UCMP system, but not involved in parallel execution of implemented algorithms, is referred to as utility software. Many of the needed utilities are provided by the manufacturers. These have been augmented by a number of special new utilities required to support a multiprocessing environment.

Implementation of the new utilities has been accomplished mainly with supplied facilities. MDS based utilities were written in PLM-80 and/or ASM-80 as the MDS's are centred on 8080 and 8085 processors, in which the utilities are executed. VAX computer resident utilities were prepared in FORTRAN and MACRO assembly language.

7.4.1 File Formats

Details with respect to the way in which files are distributed on discs or tapes, for example, are handled by operating systems. The operating system offers the programmer access to files through statements such as READ and WRITE. It is at this level, independent of particular operating systems, that formats are described here.

Files with a particular format are identified by possession of a common file name extension. Table 7.1 summarises the existing file types. Distinctive features include whether or not coded characters are employed, and the degree to which the position of information representations within the file must be defined. Coded characters (ASCII) require roughly twice the storage of non-coded information but have the advantages of being printable and more easily interpreted by human operators. Translator inputs, being prepared by a user, have fewer restrictions on format than object files, for instance, which are produced by a machine. (Note that the OBJ file name extension used does not conform to INTEL's default naming conventions where OBJ files are relocatable.)

New file types, REL and ASC, are needed for parallel implementation and transfer of information to and from the host processor. REL files result from translation through the RELOC utility. Format within REL files is similar to that used in OBJ files (INTEL(f), 1979) but load and execution addresses no longer coincide. Reliability when transferring to and from the host is enhanced using coded characters whose integrity can be checked by the user. INTEL supports such formatting in the HEX file type. However, for two reasons HEX format was found to be less than adequate as a principal mode of information storage. Firstly, no facility is provided for translation from HEX to OBJ format, and secondly, all symbolic debugging information is lost in the formation of HEX files. Hence, the

File Type	Coding	Definition of Information	Manufacturer/ Origin	File Name Extension/s
Source	ASCII	loose	INTEL/DEC	.PLM,.ASM .FOR,.MAR
Absolute Object	none	strict	INTEL	.OBJ
Load Address Relocated Object	none	strict	new	.REL
Translated for Serial Transfer	ASCII	strict	new	.ASC
Hexadecimal Object	ASCII	strict	INTEL	.HEX
Data Input to UCTS Program	ASCII	strict	DEC/user	.DAT
Executable VAX Code	none	strict	DEC	.EXE

Table 7.1: File Types Used in Operation of the UCMP System

new, more compact, and more useful, ASC representation is used.

7.4.2 MDS Resident Utilities

Facilities for the development and debugging of serial programs are provided by INTEL. These include source file preparatory aids, translators, and linkage and location tools. Debugging using in-circuit emulation is also supported. Details with respect to the input requirements and operation of these utilities are given in references (INTEL(f),(i),(j), and (k)).

7.4.2.1 Languages

All manufacturer supported languages can be employed, and combined, in software development. For the 8086 processor, these include ASM-86, the

assembly language; and three high level languages: PLM-86, PASCAL-86, and FORTRAN-86. Of these, the PASCAL-86 and FORTRAN-86 compilers execute in 8086 processors only. The Series-II MDS's (INTEL(b), 1982) which are available do not have this facility so only ASM-86 and PLM-86 are presently usable.

ASM-86 permits the most efficient possible use of processor resources while PLM-86 offers faster, and more reliable, high level code development.

All of the translators can optionally provide debugging information which is subsequently passed through object files and is of value especially when using ICE-86.

7.4.2.2 Linkage and Location

Relocatable object code, created by any of the source translators, or by previous linkage, can be combined forming a single module using the linker, LINK86 (INTEL(f), 1979). The resulting module is itself still relocatable. Library facilities (LIB86) enable management of groups of relocatable object modules.

Relocatable code is located absolutely by LOC86. Code and data are situated after location with coincident execution and load addresses.

7.4.2.3 RELOC - Load Address Relocater

The RELOC utility translates from INTEL standard absolute object files to similarly formatted REL files specially suited to operation of the UCMP system. Operating options, when running RELOC, include choice of input file, and selection of the processor in which code in the input file is eventually to be executed.

The following occur during translation:

.All load addresses below 070000H are incremented to suit code deposition at the selected processing element.

.Load addresses at and above 070000H are unaffected. This upper memory area encompasses globally accessible memory which has coincident load and execution addresses as seen by all processors.

.If an entry point for execution of code is specified within the input file, a non-maskable interrupt (NMI) vector is created. This vector eventually enables initiation of the code within the input file via operation of the control bus. When code for the master processing element is provided to operate the control bus, slave processor initiation becomes transparent to the user.

.Symbolic debugging information is preserved. As such, the original mapping between symbols and execution addresses is maintained. If ICE-86 is employed, and the absolute location of a symbolically defined object in another processing element is required, then an adjustment must be made to determine the load address ie. the location as viewed from the emulator. However, it is likely that, while debugging, references will be confined to either the local memory of the processor emulated or globally accessible memory.

.Optionally, load and execution addresses can remain coincident. In this way the start address is prepared as a NMI vector with no other effect.

.PIDATA (physically iterated data) records (INTEL(f), 1979) are translated into PEDATA (physically enumerated data) records.

The simpler, but less compact, PEDATA format is required when using the loader in 85MON.

7.4.2.4 OBJASC and ASCOBJ - File Format Transformers

These utilities allow translation from absolute object format to ASC format (OBJASC) and vice versa (ASCOBJ).

7.4.3 VAX Resident Utilities

The UCTS program is viewed by the UCMP system as a source of data. Utilities to extract and format data at appropriate points during execution are combined with the UCTS program at source level. Once placed in ASC files by CRBACK and CRASC the data can be transferred to the UCMP system using INTUCMP. Alternatively, it could be moved serially to an MDS.

7.4.3.1 UCTS Program

The serial operation and execution characteristics of the UCTS program were discussed in Chapter 2. In the present context, the program is viewed as a data generator for the UCMP hardware.

Figure 7.5 illustrates the instants during execution at which data has an appropriate state for transfer to the UCMP system. Point 'A' represents the start of execution of a new time step. Data, therefore, exists in a state suitable for execution in both the generator and network models. Alternatively, if only network solutions are of interest, point 'B' provides suitable data. As the linear equation solutions represent the most challenging area with respect to efficient multiprocessing, this point has been used most frequently in practice.

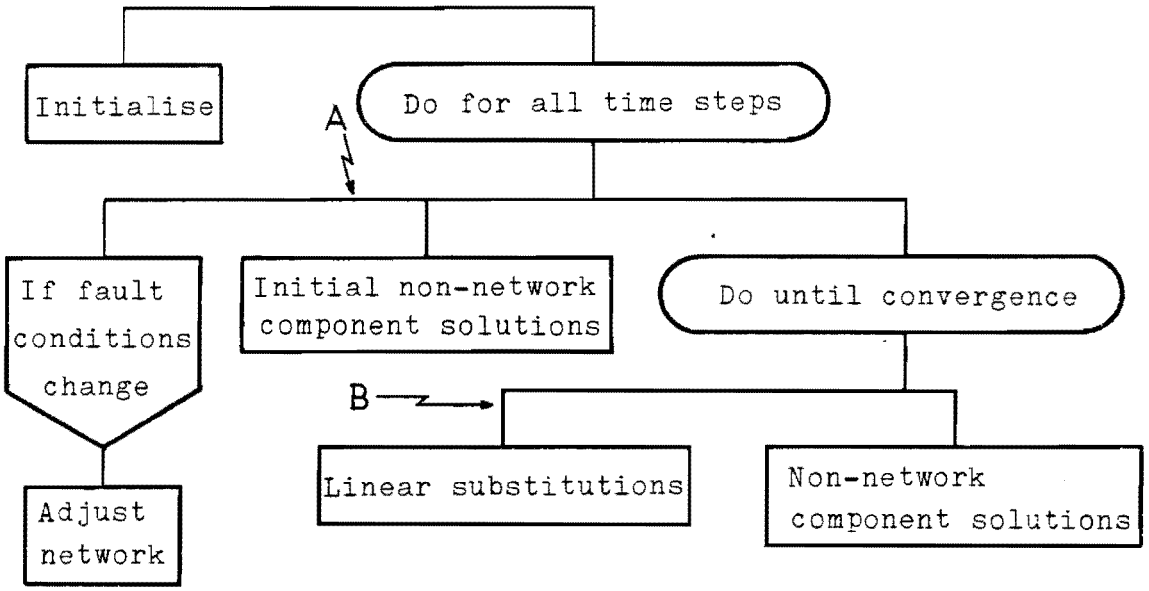


Figure 7.5: Points During Execution of the UCTS Program at which Data is in a Suitable State for Transfer to the UCMP System

7.4.3.2 CRBACK - Parallel Bifactorisation Vector Formation Utility

The approach employed during backward substitutions within the UCTS program is to make use of the same vectors as are required for the forward substitution steps. In Chapter 4 the use of further vectors especially for parallel backward substitutions was described and justified. The necessary new vectors are all derived from the forward factor matrix descriptions used in serial implementation by CRBACK.

7.4.3.3 CRASC - ASC Format File Creation Utility

CRASC provides a translation mechanism to take data during execution of the UCTS program and place it in an ASC format file. Steps involved in creating this file are:

.Because slightly different formats are used (see p.A-3 of INTEL(1) (1979) and p.8-9 of DEC(e) (1980)), real data in uncoded form is translated from DEC floating point to INTEL floating point representation. (Note that integers are stored similarly.)

.With appropriate load addresses, the resulting data is written to a file using using the PEDATA record type exclusively. Within the file, data is stored in the ASCII coded ASC form.

.Provision is made for placement of a module end record, without an entry point specification, at the end of the file.

Note that there is no automated method of ensuring that load addresses correspond to those specified in PLM-86 source files.

7.4.3.4 INTUCMP - Interactive Program

INTUCMP presents control of the 8085 processor in the host processor interface to the user via a VAX computer terminal. Selected characters are trapped and not transferred to the 8085. These are used to initiate complex operations which make use of the simple set of instructions in 85MON. Details of the available operations are given by D.G.Bailey (1981) who was involved, as a final year project student, in preparation of software for both the VAX computer and the 8085 processor. Non-interactive operations which are currently included are:

.loading of a specified ASC file to the UCMP system,

.provision of helpful information, and

.exit and return to interact with the VAX computer's operating system, VAX/VMS.

In effect, use of the INTUCMP program expands available operational choices from the very basic 85MON to all that is available under VAX/VMS including flexible file handling record management services (DEC(d), 1978). The source code for INTUCMP is written such that it can be expanded to include desired additional automated operations.

7.4.4 Stand Alone Utilities

Both the master processing element and the host processor interface include read only memory containing code enabling the respective processors to operate alone. The contents of these memories appear only in the address spaces of the respective processors, and provide some aids to debugging.

To interface to a user, evaluate performance, and control the slave elements a set of routines, which can be called by programs prepared for the master processor, are available. Those involved in communication with a terminal depend on the initialisation performed by 86MON.

7.4.4.1 8086 Monitor and Loader

The executable version of 86MON resides in a little under 4K bytes of read only memory at the top of the master processor's address space. Communication with a user is based on a serial link for which interface hardware exists enabling byte oriented transfers.

Commands interpreted by the monitor include execution controls such as 'go' and 'single step'. Other commands allow examination and updating of registers, memory and I/O ports. When an MDS is used in place of a terminal, an instruction enables loading of absolute object files if a suitable program is run on the MDS (86LOAD).

During initialisation, ie. following a reset, the terminal interface hardware is set up, and all slave processors are placed in the reset state.

As an interim facility, an instruction is included to swap from use of a terminal to communication via memory. When 85MON cooperates in transferring characters through this memory, control of the master processor becomes available to the VAX computer. Eventually, if this form

of communication is thought preferable, commands could be entered via 85MON by default. As such, only a single user interface would be needed.

7.4.4.2 8085 Monitor and Loader

85MON resides in 2K bytes of read only memory at the bottom of the host processor interface 8085's address space. Parallel data paths provide a source of commands and a return route in communication through the VAX computer.

Commands available are similar to those in 86MON, but refer to operation of the 8085. The contents of local memory within the host processor interface can only be examined using 85MON while all other RAM can be referenced by both monitors.

A command is included which allows loading of information from the VAX computer. Data arrives in PEDATA records in uncoded form ie. INTUCMP has already transformed each byte from the ASC file format. The code which performs the loading determines and sets the necessary page within MULTIBUS address space, and so is strictly oriented to its hardware environment. The lack of general applicability of this software is an unfortunate result of the limited addressing capability of the 8085.

7.4.4.3 System Oriented Service Programs

Three sets of routines can be linked to any programs prepared for the master processor enabling:

.communication with a terminal,

.control of the slave processors, and

.performance evaluation during parallel execution.

Communication routines include facilities for:

.transmission of both single characters and groups forming messages,

.reception of single characters,

.transmission and reception of byte values with ASCII coded hexadecimal representation, and

.transmission of ASCII strings expressing real numbers in decimal form, with exponent.

Slave processors have two hardware control inputs. One, common to all slaves, is a reset. The other, of which there is one for each slave, is a non-maskable interrupt. A typical control sequence would require that initially all processors were reset. Then reset would be released and all slaves execute a short sequence of initialising code culminating in each processor settling in a halt state. After this, to initiate parallel execution, NMI's would be applied to those processors whose participation is required. Finally, after execution is complete, or a malfunction has occurred, it may be desirable to again reset all slaves. Three routines enable the implementation of such control sequences by:

.resetting all slaves,

.releasing reset on all slaves, and

.applying NMI's to any desired number of slaves.

The performance of the UCMP system is measured by relating its execution speed for identical problems as the number of processors involved is varied. A basis for such measurement is provided by a clock which can be initialised, set to a predetermined state, and read at any time during

execution. Two routines are provided to enable the implementation of these three functions with the initialisation and presetting combined.

CHAPTER 8

SIMULATION OF THE PARALLEL EXECUTION OF THE SOLUTION OF LINEAR EQUATIONS

8.1 Introduction

The execution performance of a multiprocessing algorithm can be established either through application on real parallel computer hardware or via simulation. This chapter describes a simulator, called BIFSIM, which models execution of the PBIF, parallel bifactorisation, algorithm. Performance can be determined over a wide range of conditions. BIFSIM itself is a program, written in FORTRAN, and run on the VAX computer.

The objective of BIFSIM is to accurately portray the performance expected of real systems. As such, simple models like those used by Wing and Huang (1980) are inadequate because management related processing and the effect of non-constant task execution times are ignored. As the PBIF algorithm dynamically allocates tasks, small changes in execution times can dramatically alter execution sequence. Therefore, varying operation execution times are modelled. In addition, very detailed models of management tasks are used.

During execution on real multiprocessing hardware, it is not possible to measure individual overheads without introducing distorting delays in execution. A simulation, on the other hand, can accurately measure separate overheads with no effect on simulated performance. In BIFSIM advantage is taken of this possibility and the relative effect of four overhead categories are recorded.

Results presented are based around a set of benchmarks varying in both processor complexity and linear network form. Selection criteria included both practicality and a need for comparison with the results of other researchers.

8.2 Principle of Operation

The environment under which a user can control operation of the simulator and prepare interpretable results is illustrated in figure 8.1. The BIFSIM program models the operation of a multiprocessor step by step throughout the execution of linear substitutions.

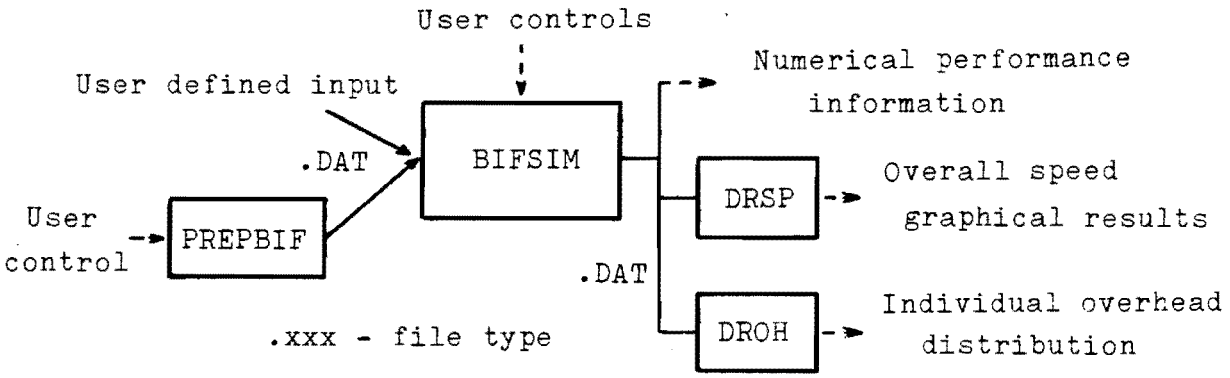


Figure 8.1: Execution Environment and Options in Use of the Simulation Program

A separate program, PREPBIF, is used to generate network descriptions. Random element distribution is assumed. Controls are provided to enable variation of system size and sparsity. Sensitivity to these parameters can then be analysed as small variations are easily implemented. Such is not the case when analysis is limited to real power system data of which relatively few sets are available. An ill-conditioning problem, which was not identified by the simulator, is discussed in Chapter 9. The problem is attributable to unfortunate element distribution originating from the form of some real power systems.

Results are written to a file which can be read and, if desired, immediately interpreted. Alternatively, the file can be delivered, unaltered, to further programs (DROH and DRSP) which prepare graphical descriptions of both overall system performance and the relative effect of individual overheads as a function of the number of processors.

Figure 8.2 illustrates the operation of the BIFSIM simulator. With reference to the modelled execution, instants at which tasks are completed are identified. At these instants idle processors are assigned any ready tasks, appropriate time base variables are incremented, and any overheads are recorded. The simulator, therefore, models performance over a period at each of many instants. The simulated period between these instants is determined by the modelled task execution times.

Before assignment to a processor, each task is alloted an execution time selected from a defined distribution. In this way, the varying instruction execution times of processors, eg. data dependent arithmetic operations, can be accurately modelled. Tasks are categorised and each type can be assigned a different distribution. Provision is made to vary the average, the width, and the form of distributions used. In practice a rectangular distribution has been employed most frequently.

8.3 Overheads

An overhead is any event resulting in a difference between the speed of a parallel processor and the product of the speed of a single processor and the number of processors.

8.3.1 General Categories

For any multiprocessor, there are three areas in which overheads can arise. They are categorised here as:

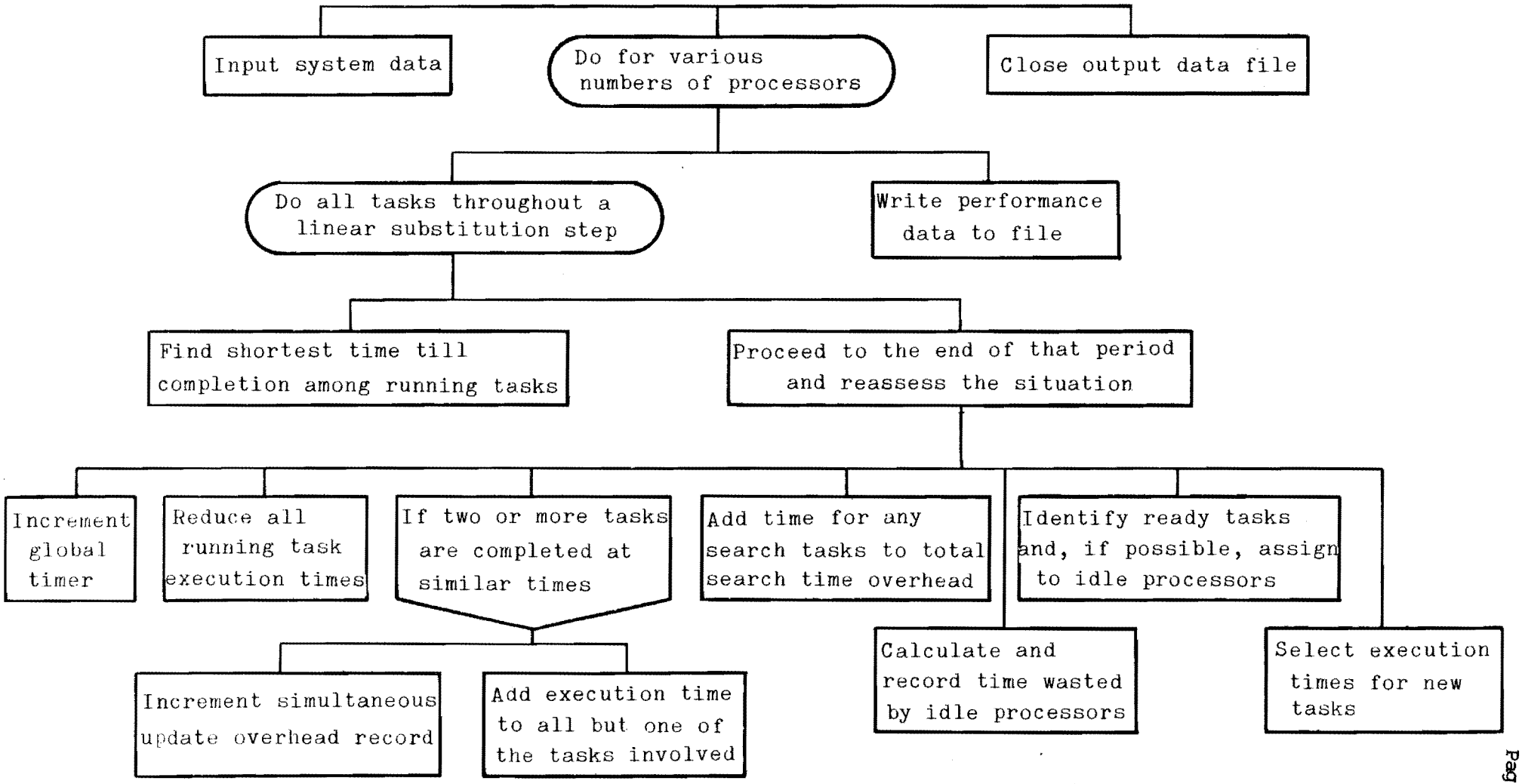


Figure 8.2: Execution Sequence of the BIFSIM Simulator

(a)-algorithm modifications,

(b)-management processing, and

(c)-processor idling.

Changes to methods employed in reaching solutions can lessen the level of overhead provided by categories (b) and (c). That is, changes from an 'optimal' serial approach may be valuable in forming greater independence between tasks or in other ways allowing more efficient distribution of tasks among processors. In implementing such changes, the total amount of processing required cannot be reduced and, in general, increases. This increase, best measured by comparison of serial execution times, forms the algorithm modification overhead-(a).

During the execution of programs in a multiprocessor, some portion of execution must be devoted to tasks which do not occur in serial solutions. Tasks in this category include facilities distributing useful tasks to appropriate processors, and supervision of synchronisation between processors. Time spent in execution of these tasks forms the management processing overhead-(b).

The idling overhead, (c), arises when processors cannot continue operation and must wait for the completion of an operation by other processors. Idling can occur for two reasons:

.no tasks are available due to an insufficient supply of independent tasks, or

.a resource, common to more than one processor, is busy.

8.3.2 PBIF Algorithm

Overheads from all categories can affect the execution of the PBIF algorithm:

.Algorithmic changes, in the form of special reordering of network matrices, can result in increased task independence eg. formation of blocks along the diagonal. Such changes result in increases in the total number of elements in factor matrices, and consequently in more processing.

.Many operations by all processors are specifically related to ensuring correct operation in a multiprocessing environment. Searches for ready tasks, and semaphore setting and resetting are examples.

.It is likely that at points during execution processors will wait for ready tasks because none are available. In addition, processors will attempt to utilise busy common resources which could be either software or hardware based. Common software resources include semaphore protected memory elements. Common hardware resources include the network which interconnects processors and globally accessible memory.

8.3.3 Models Included

Overheads from categories (b) and (c) only are modelled ie. it is assumed that no special reordering is applied in the formation of factor matrices. A design objective in development of the PBIF algorithm was avoidance of block formation using reordering techniques to avoid fill-in. However, it has not been determined that all reordering possibilities are of no value.

Management processing overheads modelled include:

.Search for ready tasks ie. vacant columns,

.Setting of semaphores during the search,

.Setting of semaphores during update operations, and

.Decrementing, with semaphore protection, elements of the vector which dynamically records the number of elements in each row which are not yet substituted.

Overheads due to processor idling include:

.No tasks available, and

.An element is presently being updated by another processor.

An overhead which is known to not be simulated is processor idling due to coincident requests for use of common hardware resources ie. the network interconnecting processors and globally accessible memory. It is possible that further overhead types have not been identified and have consequently been ignored. However, a comparison between real hardware performance and simulated performance in Chapter 9 concludes that there is no appreciable inaccuracy in the simulation resulting from unidentified sources of overhead.

Common hardware resources, and the network through which they are accessed, vary depending on the multiprocessor in question. For a particular system, the overhead can be quantified and included in the simulation. The objective of the simulator developed, however, is to illustrate the value of the PBIF algorithm independent of hardware constraints. Results of the simulation, therefore, must be qualified by consideration of available or future hardware implementations. The likely

effect of the overhead for the UCMP system was investigated in Chapter 6 concluding that, for the number of processors invisaged, the overhead is insignificant. As shown in Chapter 5 there is much scope for increase in the complexity of hardware interconnecting networks aimed at increasing data traffic efficiency. Such systems could be of value when using more and/or faster processors.

8.3.4 Categories for Recording

To enable comparison of the relative effects of the various overheads, provision is made for individual measurement. The overheads are categorised as :

1. No tasks available
2. Search for vacant columns, including semaphore updates
3. Decrementing Dynin including semaphore updates
4. Simultaneous updates of elements of the z vector

In the context of the PBIF algorithm, each processor is constrained to search among a limited number of columns for a ready task. In practice this will be an efficient approach as the most likely tasks to be ready are usually nearest to those just completed. On reaching the limit the processor reinitiates the search at its original starting point which may, by this stage, contain a ready task.

8.4 Benchmark Processor and Network Choice

Many systems varying in size and sparsity, all generated by the PREPBIF program, were used. Of these two were selected for presentation of quantitative results. Although there is nothing radically different about these systems, they are considered valuable because firstly, they have form

which is typical of power systems, and secondly, they are similar to benchmarks selected by other researchers.

The networks used were:-

.a 400 bus network with randomly distributed elements, 99.2% sparse (similar to that used by Wing and Huang (1980))

.a 2000 bus network with randomly distributed elements, 99.5% sparse

The processors modelled were:-

.a 16-bit microprocessor (INTEL 8086)

.a 48-bit mainframe (BURROUGHS B6700)

These were selected as their speeds are expected to give results indicative of two important and separate classes of processor. The execution rate data used was that given in table 6.1. No account has been taken of the available bus structures for the two processor types.

8.5 Performance Measured

Three sets of simulated performance characteristics are presented. In all cases performance is expressed as a function of the number of processors. The first set (figures 8.3 and 8.4) illustrate overall performance in terms of speed relative to one processor. Measures of the distribution of the effect of individual overheads for both processor types when using the 400 bus network are given in the second set (figures 8.5 and 8.6) and those for the 2000 bus system in figures 8.7 and 8.8. Overheads are identified using the categories defined in section 8.3.4.

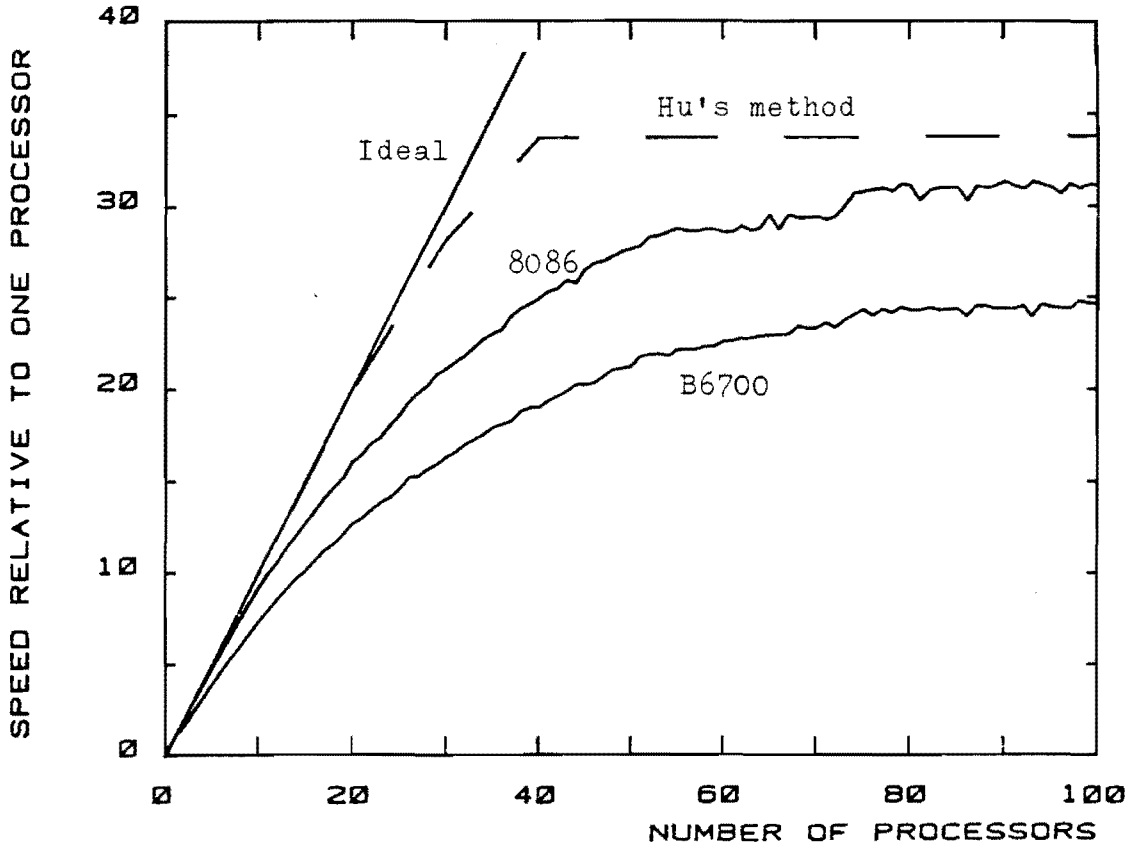


Figure 8.3: Performance Comparison for the 400 Bus Network

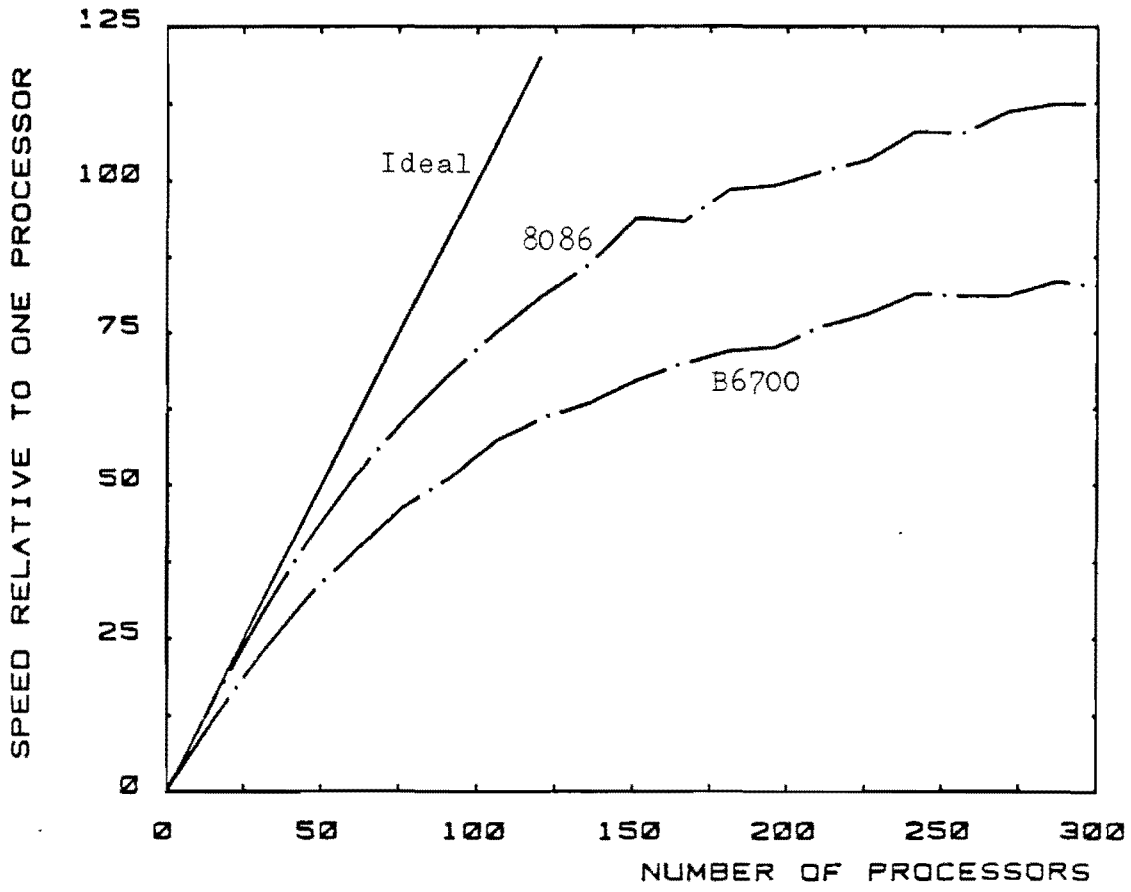


Figure 8.4: Performance Comparison for the 2000 Bus Network

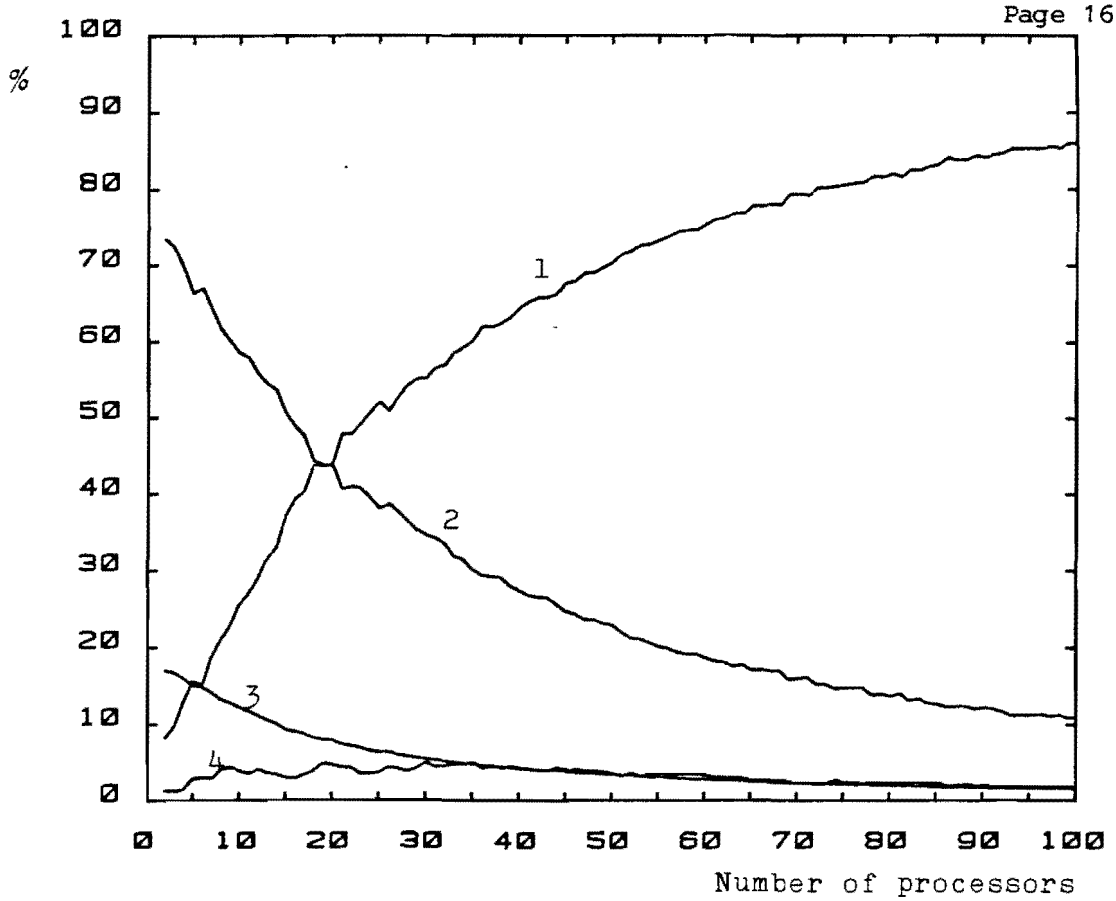


Figure 8.5: Relative Time Spent in the Execution of Individual Overheads (B6700 Processor and 400 Bus Network)

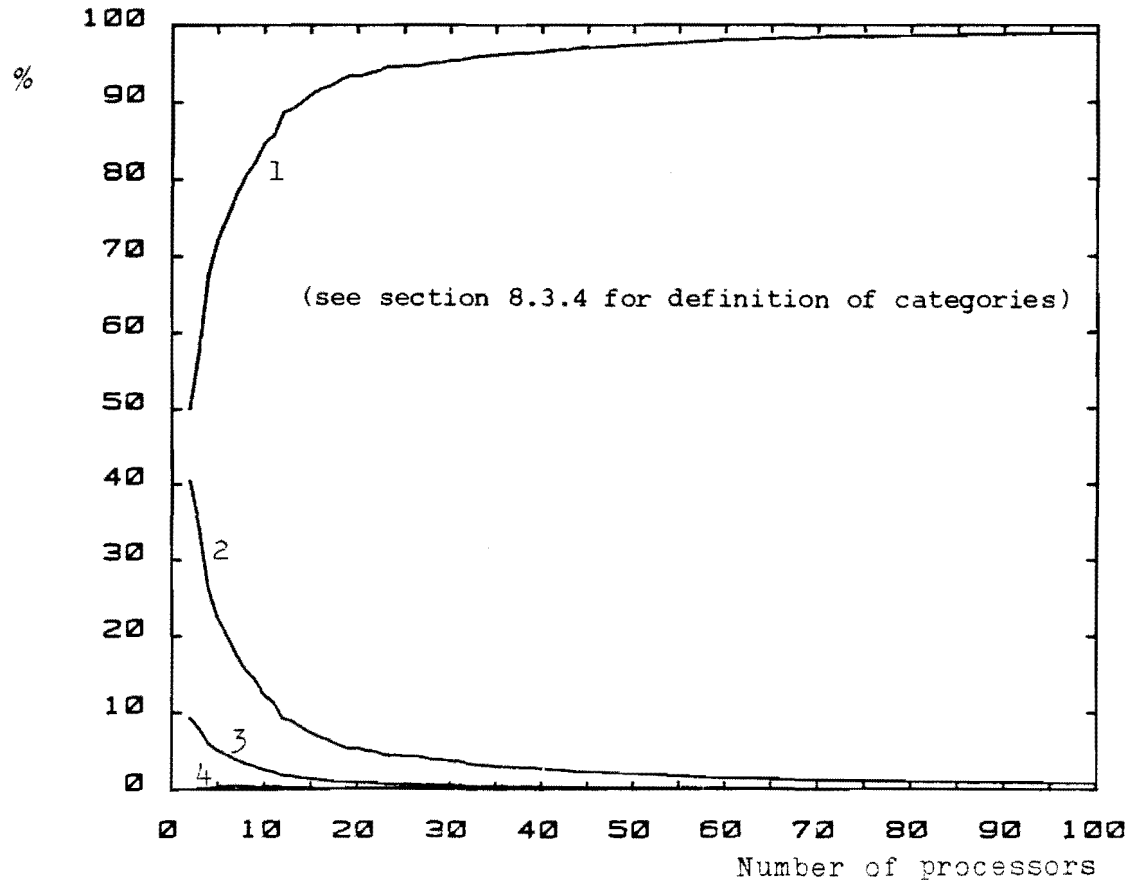


Figure 8.6: Relative Time Spent in the Execution of Individual Overheads (8086 Processor and 400 Bus Network)

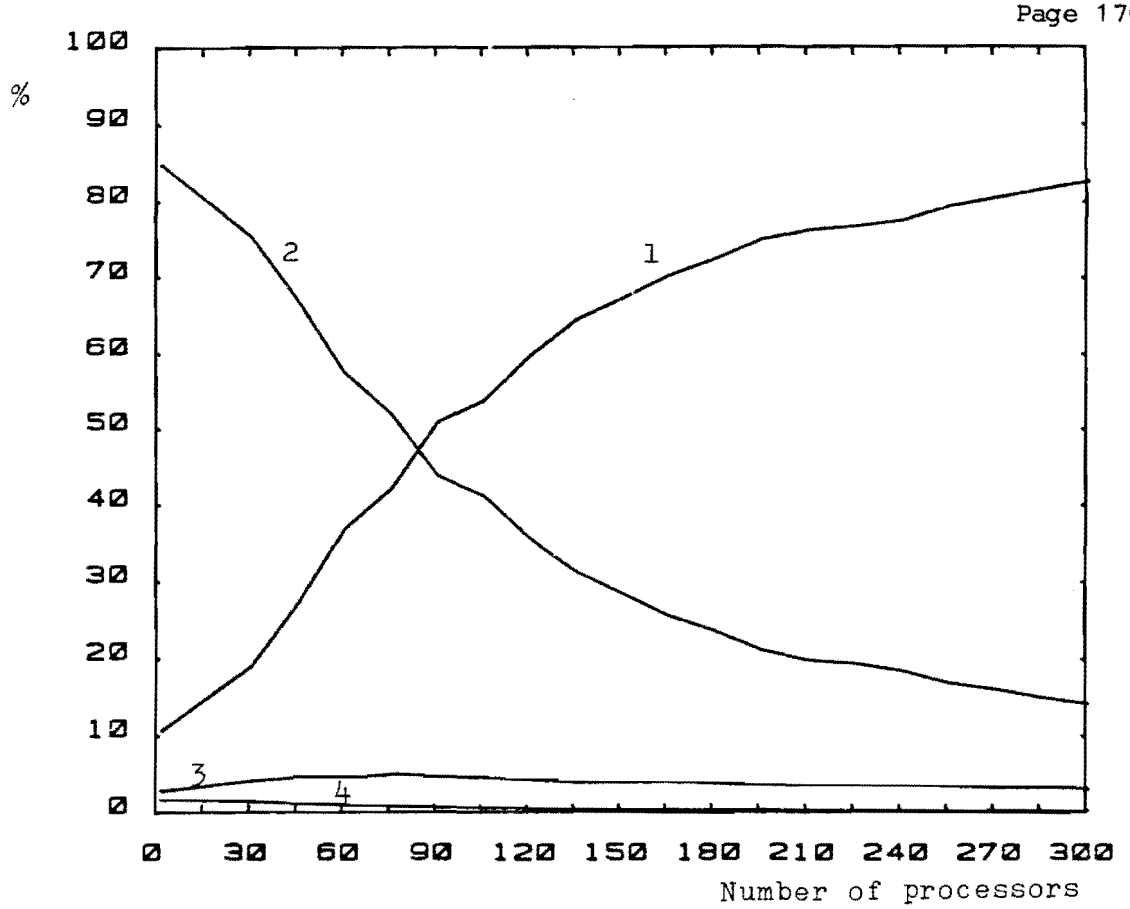


Figure 8.7: Relative Time Spent in the Execution of Individual Overheads (B6700 Processor and 2000 Bus Network)

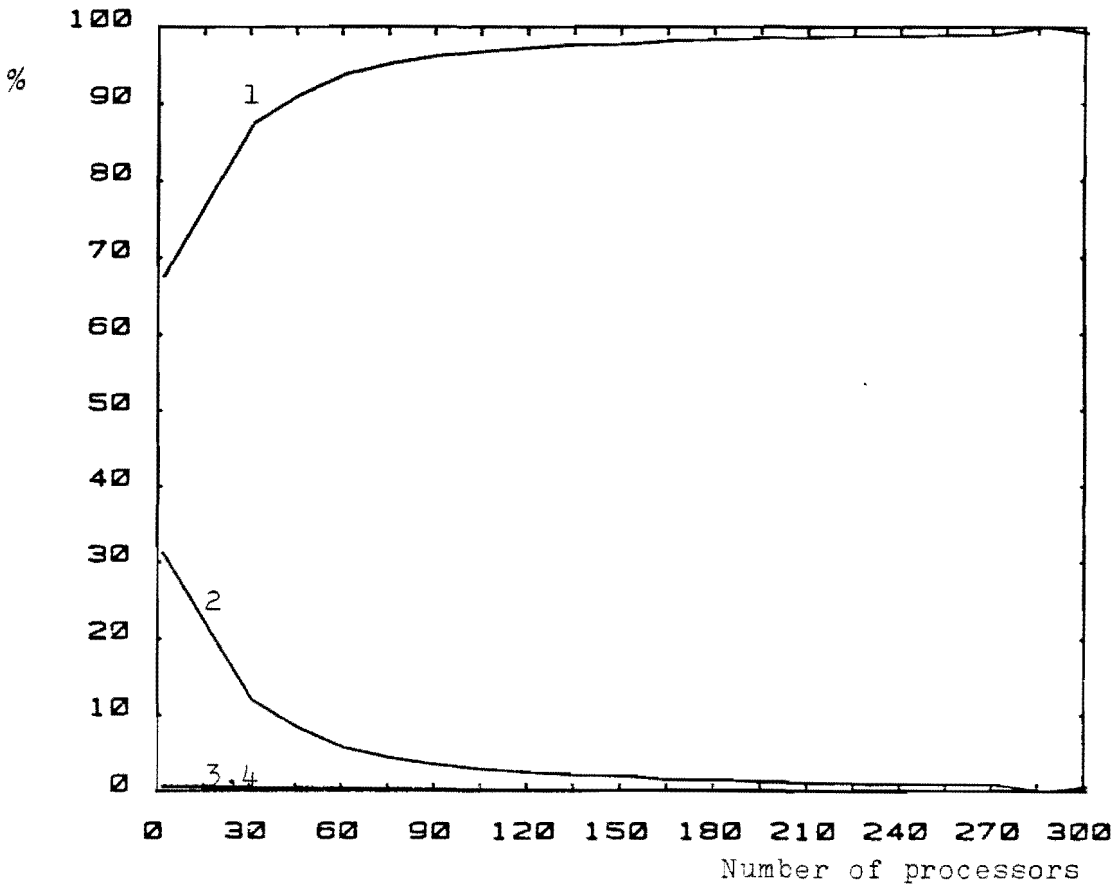


Figure 8.8: Relative Time Spent in the Execution of Individual Overheads (8086 Processor and 2000 Bus Network)

8.5.1 Overall System Performance

Speed up measures for the two processor types are illustrated and compared in figures 8.3 and 8.4. In obtaining these, single processor operation was not optimised and the execution sequence defined by the PBIF algorithm was used*. Note that the maximum effective number of processors when using the 8086 processor is around 30 with the 400 bus network and over 100 for the 2000 bus system. Because it has a proportionately higher management overhead, the B6700 performance is consistently below that of the microprocessor.

Changing the variance of the speed data was found to have no observable effect on the position of the performance curves. It was also found that performance is insensitive to the sparseness of the matrices involved.

In assessing the effectiveness of an algorithm, an accurate indication of the absolute limits to performance is valuable. For the substitution steps in linear solutions such a 'constrained ideal' characteristic is provided by simple simulation eg. as described by Wing and Huang (1980). This uses Hu's method, an approach which fully exploits available parallelism, but management overheads are not considered. The constrained ideal performance cannot be achieved practically. However, the degree to which the ideal is approached provides a measure of the algorithm's effectiveness. Included in figure 8.3 is a curve corresponding to the results obtained using Hu's algorithm for a similar network.

---

* - The effect of this constraint has been determined, by examination of the code involved, to introduce errors in execution time of less than 2% and less than 10% for the 8086 and B6700 respectively.

8.5.2 Distribution of Inefficiency Among Overheads

The objective of the PBIF algorithm is to obtain maximum possible execution speed. The number of processors, or more importantly the ratio of the number of processors to the size of the network simulated, was not specified at the design stage. Therefore, the PBIF algorithm was heuristically aimed at good performance over a wide range of number of processors. Different algorithms are likely to result in different distributions of performance over a range of the number of processors. Measurements of the distribution of efficiency among overheads in execution of the PBIF algorithm provide a basis for tuning or selection of the best algorithm when the network modelled, and the number and configuration of processors is specified.

Taking the 400 bus, 8086 based performance (figure 8.5), for example, consider high numbers of processors initially. Management overheads are less significant than processor idling. This suggests an increase in management may be valuable. Alternatively, considering low processor numbers, management overheads dominate. Consequently, a reduction in management processing (accompanied by reduced exploitaion of parallelism) may increase speed. Between these extremes, at a point around 25 processor operation, both overhead categories occur in similar portions. Processing efficiency at this point is about 70%.

Comparing the overhead levels for both processor types, the overhead due to management is greater for the faster processor. The reason is as follows. Instructions to be executed include firstly, useful processing and, secondly, management processing. The former predominantly involves the execution of floating point instructions, while the latter requires integer related and logical instructions. Relative to the microprocessor, the mainframe processor is much faster at executing floating-point

instructions but similar in speed for other instructions. Hence, the mainframe spends a greater proportion of its time in management type execution.

8.6 Conclusions

A program called BIFSIM, and results of its execution, have been described. The program is intended to simulate the operation of a multiprocessor during execution of the implemented PBIF algorithm. Validation of the operation of BIFSIM, through comparison with real parallel executions, has not been presented. However, the detail included in the models used provides a basis for very accurate estimation of performance.

The modelled execution performance suggests that the speed up is sufficient to greatly reduce the bottleneck arising in transient stability analysis programs due to the solution of linear equations.

CHAPTER 9

HARDWARE PERFORMANCE - THE SOLUTION OF LINEAR EQUATIONS

9.1 Introduction

In this chapter implementation and testing of the Parallel Bifactorisation algorithm, using the UCMP multiprocessor, are described.

Comparison is made with the performance predicted by the BIFSIM execution simulator. Confidence in the accuracy of the simulation is thereby increased. The small useful range of the UCMP hardware can then be combined with the virtually unlimited range of the simulator to confidently ascertain the full capabilities of the PBIF algorithm.

A number of sets of real power system data are used to examine likely performance in possible typical situations. An interesting problem, which results in performance degradation to an extent which differs from network to network, is identified in the course of this study. The reasons for its existence are examined, and possible solutions are mentioned.

Two parameters in the implemented PBIF algorithm are identified as suitable for tuning to enhance performance. Attempts to find optimal values are described and the importance of appropriate choice in likely real systems is discussed.

In any multiprocessing implementation, consideration of bus conflict problems is important. The UCMP system was designed for implementation of the PBIF algorithm and to have insignificant overhead due to conflict. Large numbers of processors are modelled to establish the validity of the design approach and the way in which bus saturation is reached.

Inaccuracies due to necessary assumptions in these validation tests are also considered.

9.2 Program Implementation

The executable code requirements of the PBIF algorithm were realized using just two source programs: one for the master processor, and the other providing identical code for each of the slaves. Both were written predominantly in PLM-86. Some small routines, such as variable length semaphore setting, were prepared using ASM-86.

The single data segment automatically produced by the PLM-86 compiler is used for local data references. Globally accessible variables are individually defined in set locations ensuring coincident addressing by master and slaves.

The linear solution steps are a subset of the complete implementation which includes non-network models. The complete master program resides in about 25K bytes of local memory, while the slave program requires only 17K bytes.

9.3 Comparison of Performance of Hardware with that Predicted by Simulation

To confidently apply the BIFSIM simulator to ascertain performance of the PBIF algorithm over a wide range of conditions, comparison with real hardware, even if only over a limited range, is invaluable. The UCMP system provides an opportunity for such a comparison.

The choice of suitable benchmark network data in comparing real and simulated operation is important. The network chosen was selected to match the capabilities of the UCMP hardware, and, at the same time, have form typical of power systems. Matching here means, firstly, that operation

does not become saturated well before the maximum number of processors is reached, and secondly, that appreciable drops in effective processor utilisation do occur. The choice made, to use a 30 bus system with sparsity coefficient of 0.83, was based on experience with the simulator which suggested saturation would occur at around 8 to 10 processor operation. The system was generated using PREPBIF.

Performance characteristics of both simulated and real operation are compared in figure 9.1. Clearly, the form of both is very similar, and the absolute difference between the characteristics is very small. For instance, the difference for six and seven processors is about 3% and 5% respectively. Observation of figure 9.1 shows that this order of error is less than that arising in the simulator (due to factors such as random operation execution time selection). Hence, it is concluded that the simulation provides performance estimates indicative of real system implementation. A corollary to this conclusion is that no significant overhead has been neglected in the model used in the BIFSIM program. The idling overhead due to simultaneous requirements for hardware resources is not simulated, but no error is introduced because it has been shown (section 6.2.2.2) that this overhead is insignificant for the UCMP system executing the implemented PBIF algorithm.

Therefore, two factors lead to the conclusion that the simulator will produce a reliable performance indication over a range of conditions (for the 8086 processor). Firstly, it is known that detailed overhead and realistic process modelling are included, and secondly, a very close similarity between modelled and actual performance has been illustrated. The deliberately excluded idling overhead can and should be considered when particular hardware implementations are selected.

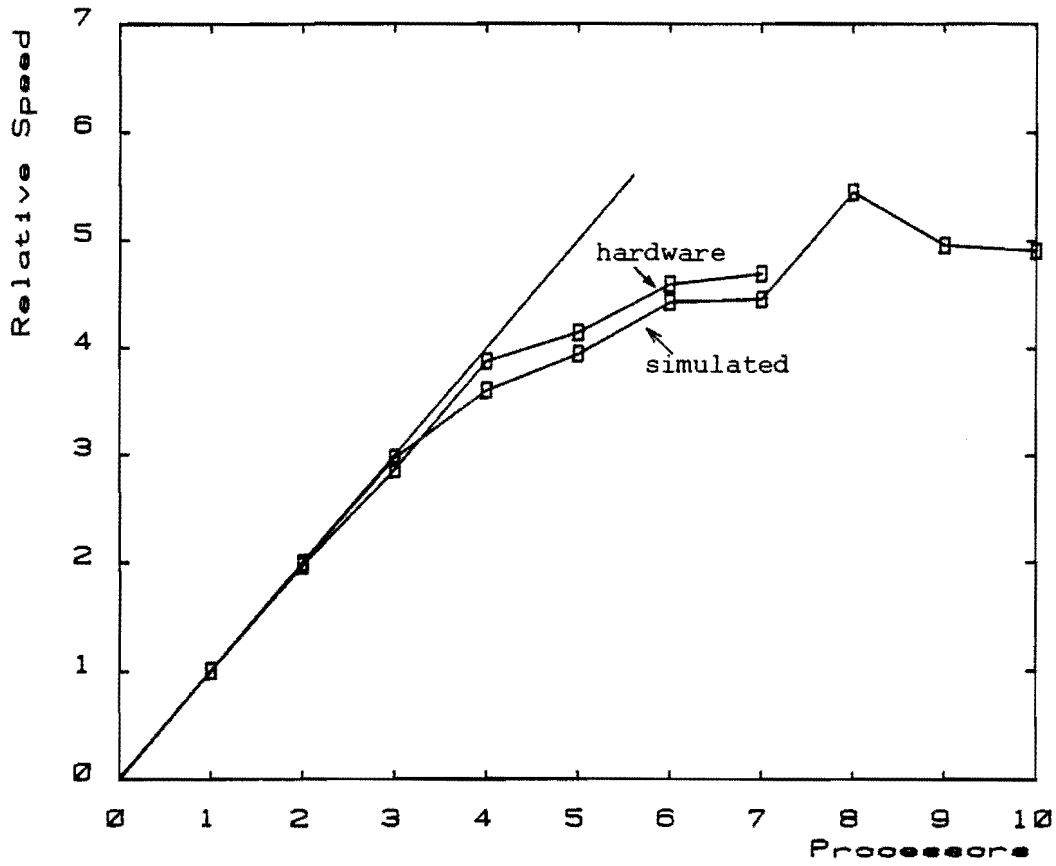


Figure 9.1: Comparison of Real and Simulated Performance (using a 30 Bus test system)

9.4 Various Practical Power Systems Modelled

Performance measurements, using five sets of real power system data, were made to enable comparison of characteristics. The resulting characteristics are both different and interesting when compared with the performance expected through experience with the simulator.

Table 9.1 summarises the form of each of the data sets employed. All were prepared via the UCTS program. In this chapter systems are referred to by their number of buses.

Figures 9.2 to 9.6 illustrate the measured performance characteristics of all systems.

System Name	Number of Buses	Sparsity Coefficient
3-bus	3	22.2%
8-bus	8	65.6%
24-bus	24	87.5%
35-bus	35	93.75%
205-bus	205	98.1%

Table 9.1: Networks Used in Hardware Performance Tests

The speed up achieved in all cases is less than that predicted by simulation of randomly distributed networks with similar size and sparsity. The most graphic example of poor performance is the eight bus network where no improvement is measured when operating second and subsequent processors. Note that better results were achieved with the three bus system.

When the exact network forms of the real systems were applied in simulation, the same poor performance characteristics resulted. Therefore, it was concluded that a problem exists, which is related to the form of networks, and varies greatly from network to network.

9.5 Nodal Distribution Ill-Conditioning

The performance degradation, uncovered by observation of the characteristics measured in the previous section, is due to ill-conditioning related to task dependence and results from unfortunate network element distribution. The problem, which will be called nodal distribution ill-conditioning, can be related to origins in both network layout and choice of ordering scheme.

The physical origins of task dependence can often be observed in problem specifications. An example is in the formation of groups of

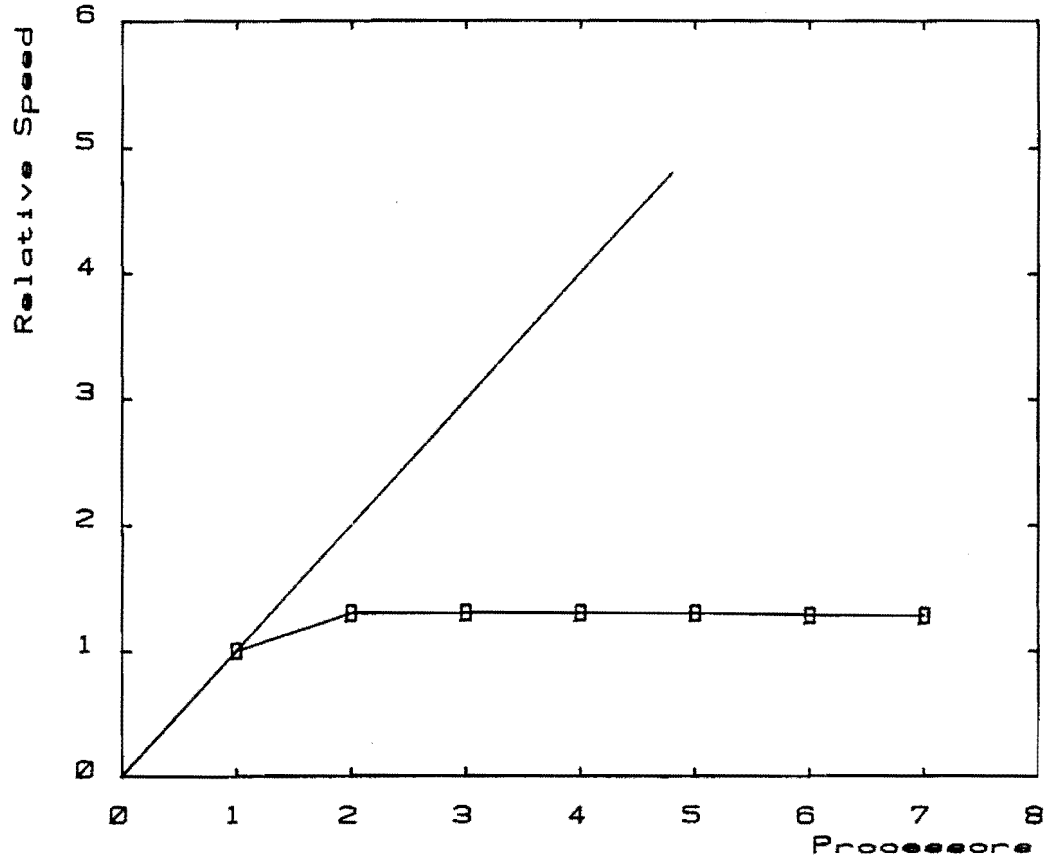


Figure 9.2: Hardware Performance - 3 Bus Network

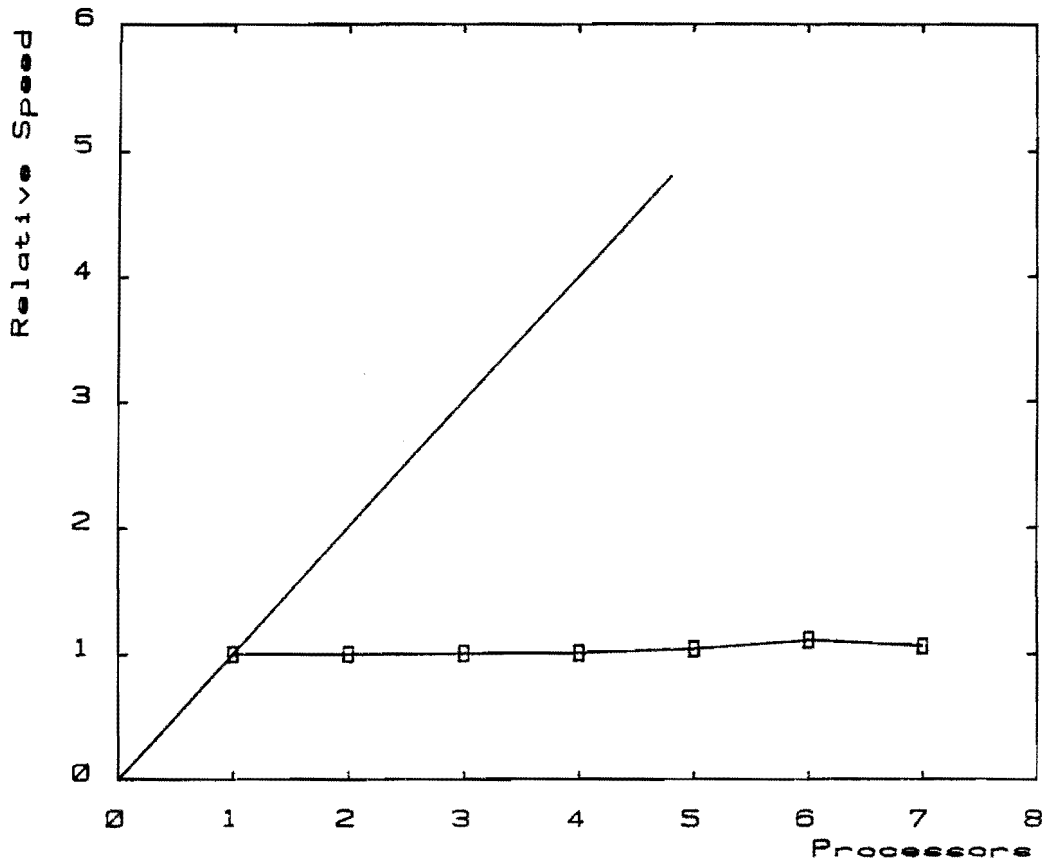


Figure 9.3: Hardware Performance - 8 Bus Network

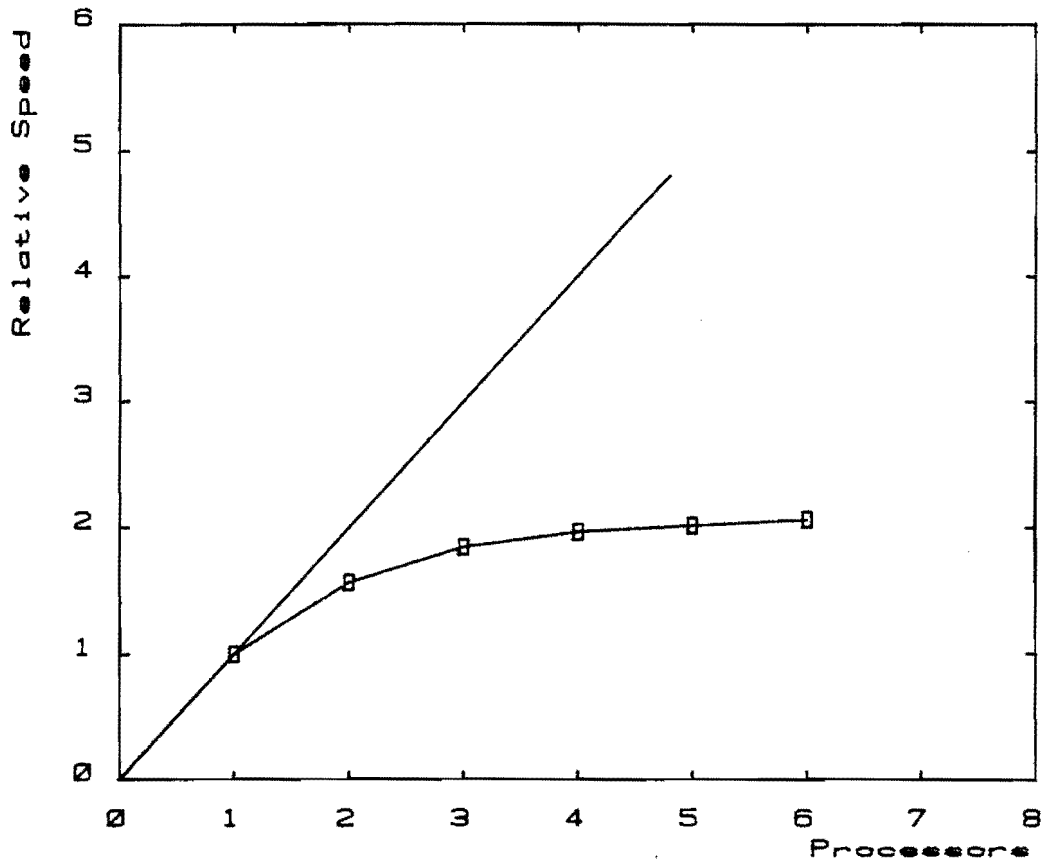


Figure 9.4: Hardware Performance - 24 Bus Network

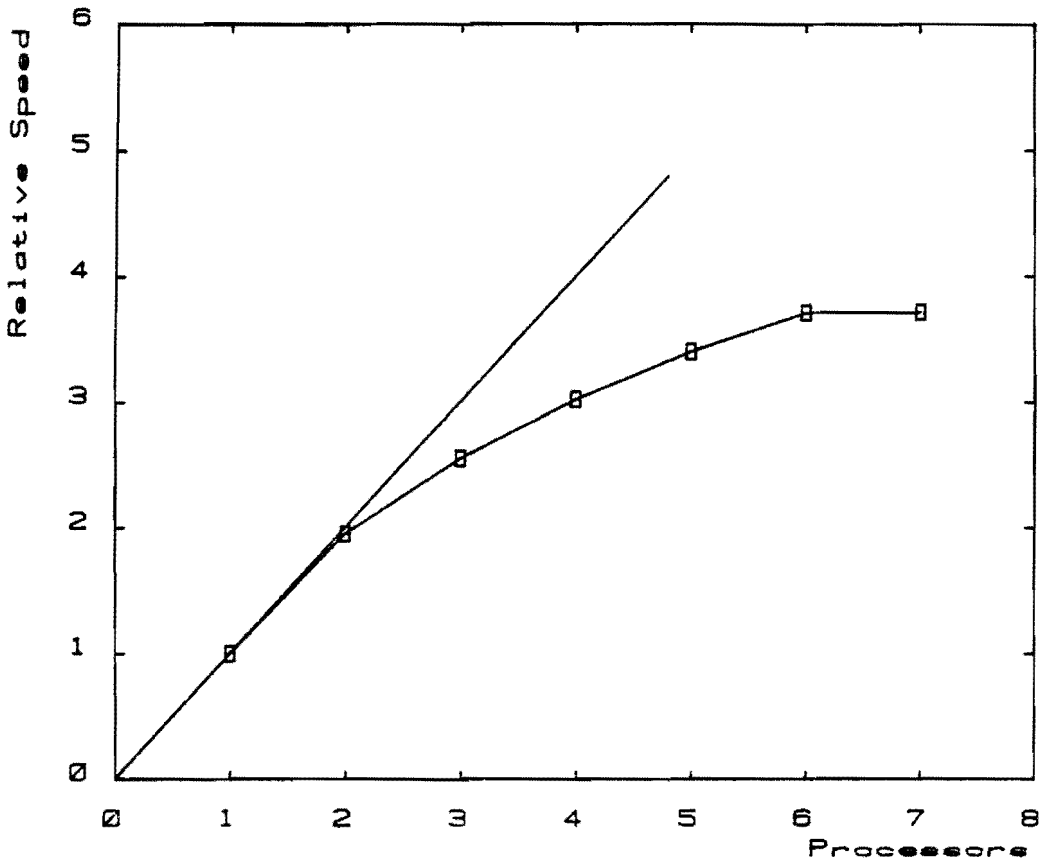


Figure 9.5: Hardware Performance - 35 Bus Network

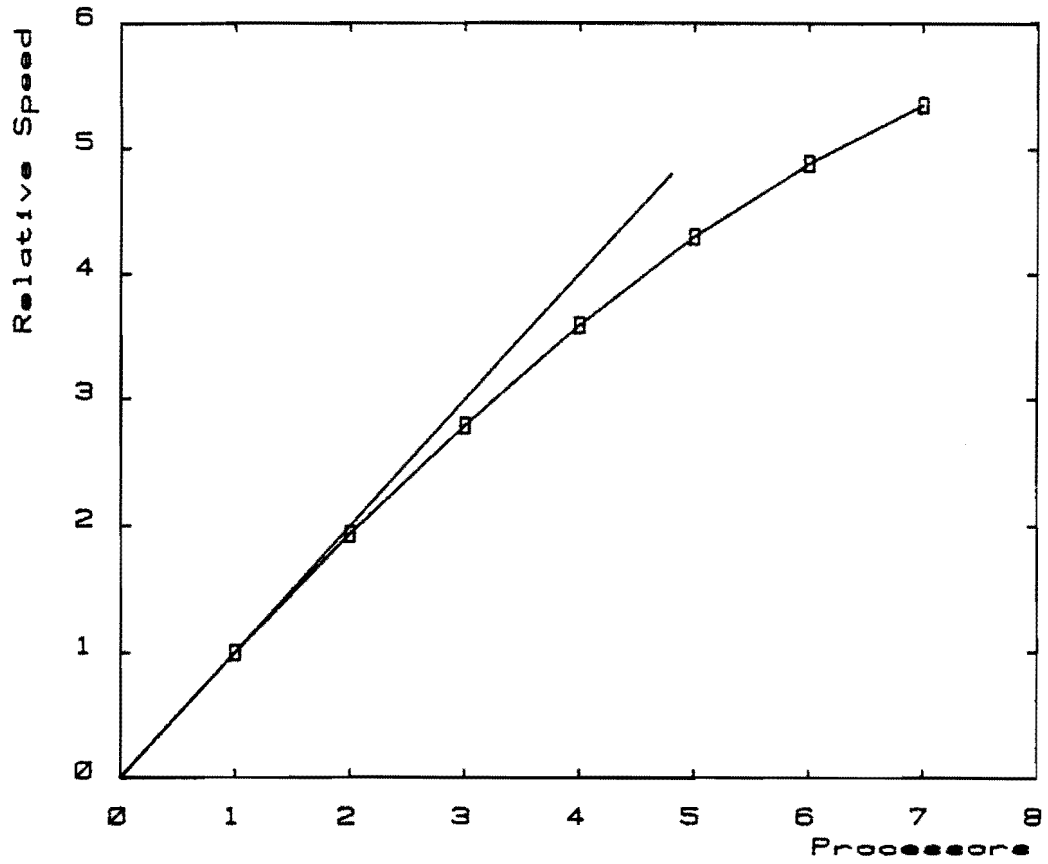


Figure 9.6: Hardware Performance - 205 Bus Network

independent tasks in block approaches to linear substitution (Chapter 4). If power systems can be divided into almost independent groups of nodes the cut set will be small and parallel execution will be efficient. The simultaneous fill-in due to deviation from serial 'optimal' ordering can be justified up to a limit.

Nodal distribution ill-conditioning results from serial element connection in networks. Consider the two networks depicted in figures 9.7 and 9.8. Network 1 has a random selection of bus interconnections, and network 2 has a single line along which nodes are situated. Also given in figures 9.7 and 9.8 are the original network and combined factor matrices. Both systems are reordered using the 'dynamic least number of connections'

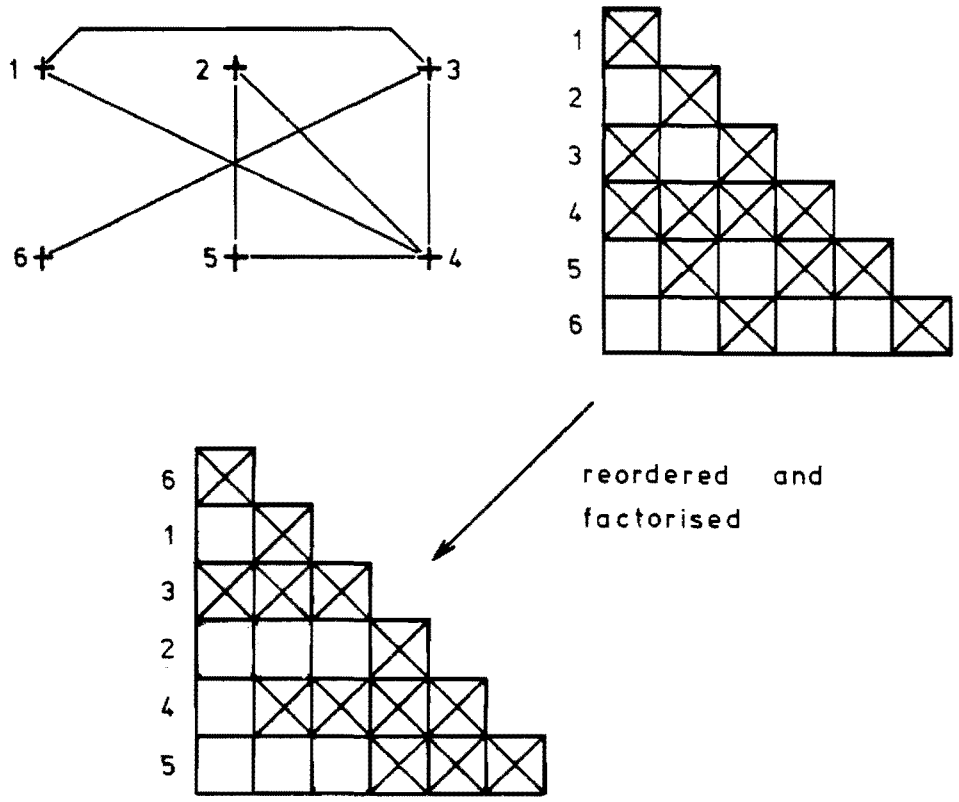


Figure 9.7: Example Network 1 - Random Distribution

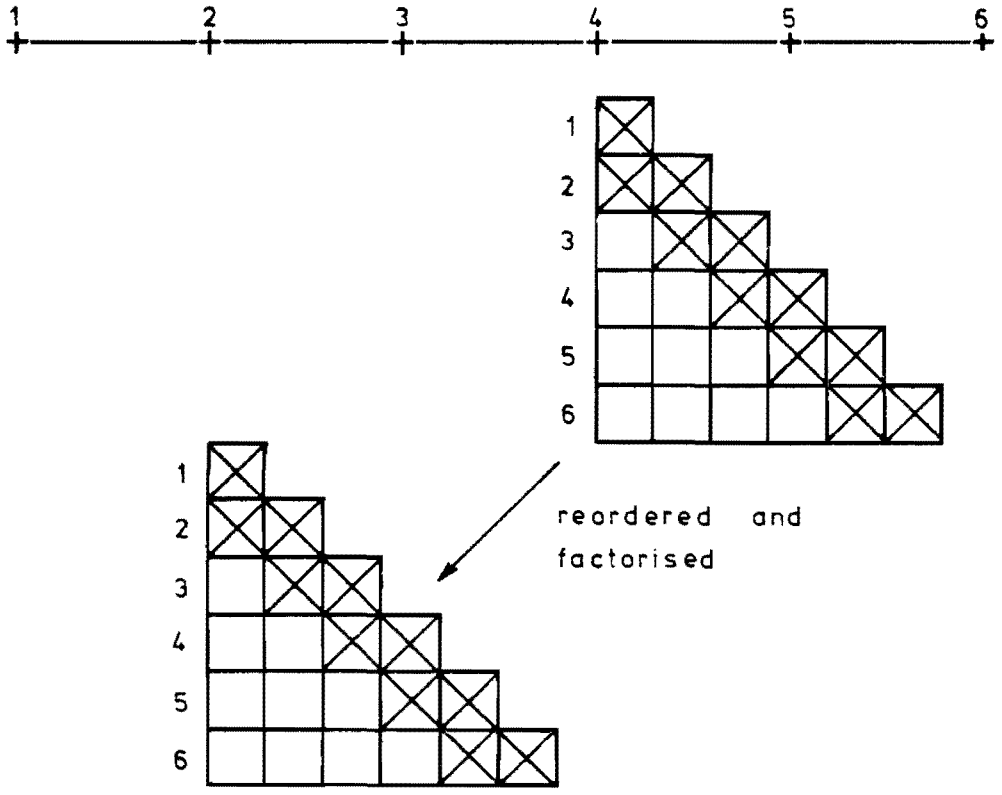


Figure 9.8: Example Network 2 - Serial Distribution

strategy (Zollenkopf, 1971) which is used in the UCTS program. In both cases there is no fill-in. While constrained to the PBIF algorithmic approach, task graphs illustrating execution possibilities are given in figure 9.9. As the vertical axis is indicative of execution time, it is clear that far more scope for efficient parallel execution exists for the randomly distributed system.

Because of the serial nature of processing they imply, the elements situated beside or close to the diagonal are identified as being responsible for nodal distribution ill-conditioning. The same problem, in its worst possible form, was faced by Alvarado (1979) while trying to cope with a number of integration time steps simultaneously. Element distribution, likely to contribute to the problem, is observable in figure 9.10 which is the combined factor matrix for the 24 bus test system. In practice, this resulted in mild performance degradation.

Performance can be improved by reordering. For example, if network 2 is reordered as in figure 9.11, the task graph, also in figure 9.11, shows the increased scope for parallel execution. The improvement is reached at the expense of introducing a single element of fill-in. The reordering used corresponds to the formation of two blocks ie. nodes (1 and 2), and nodes (5 and 6). It is therefore possible that block approaches to ordering may combine effectively with the PBIF algorithm in solving practical problems.

Optimal choice of ordering scheme is very difficult. For serial implementations it is the result of a trade-off between three factors (Brameller et al, 1976): maintenance of high numerical accuracy, preservation of sparsity, and maximum computational efficiency (taking account of the computation required to implement the ordering). In parallel applications these must be further complicated by the new, and

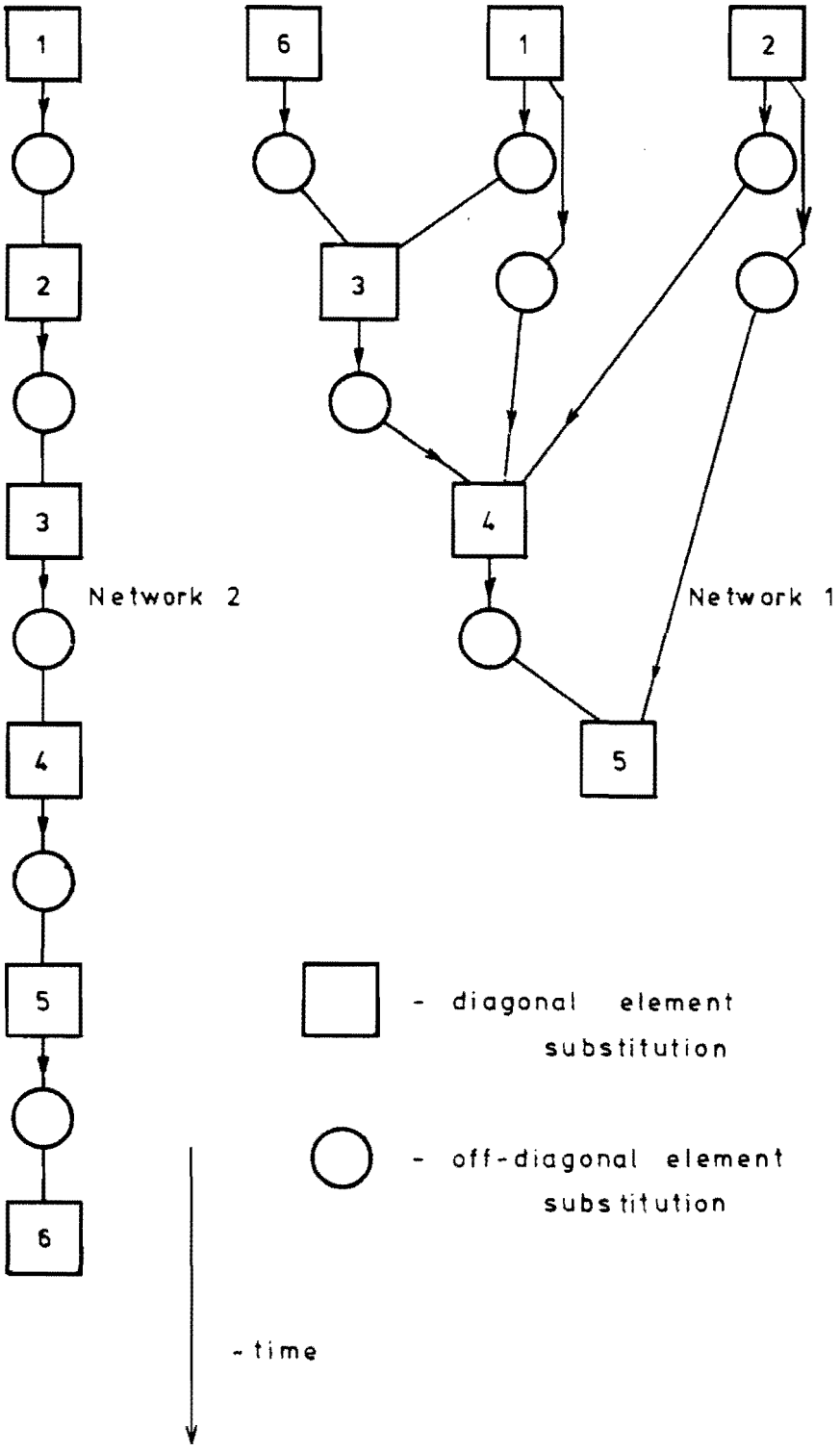


Figure 9.9: Comparison of Task Dependence for Networks 1 and 2

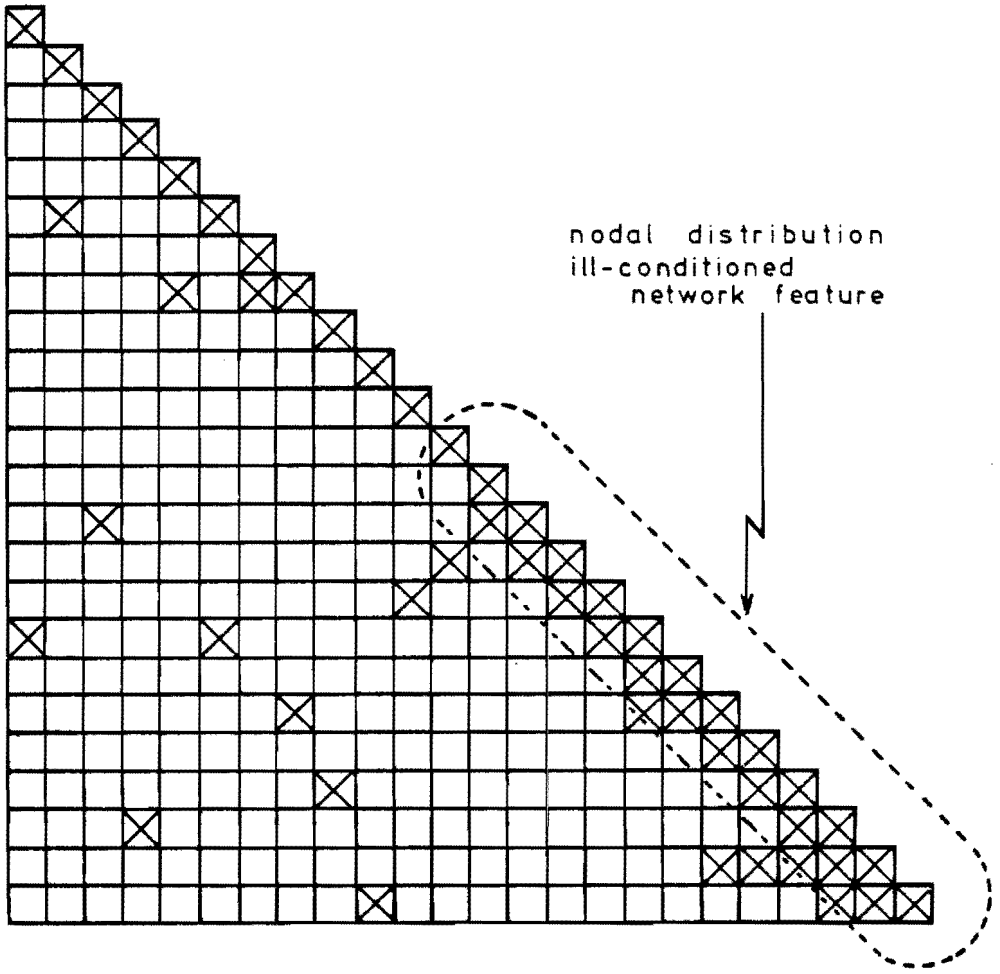


Figure 9.10: 24 Bus System Combined Factor Matrix Element Distribution

conflicting need, to avoid positioning of elements near the diagonal in factor matrices. Even worse, the weight given to this need will vary depending on the number of processors involved.

9.6 Variations and Sensitivity to Program Parameters

Two parameters, within the implemented PBIF algorithm, can be optimised as a result of trade offs between factors involving network form and multiprocessor structure. Both were discussed in Chapter 4 while introducing the PBIF algorithm. The first is a limit to the number of tasks searched for ready status, and the second is the length of a delay imposed after each check of semaphore status.

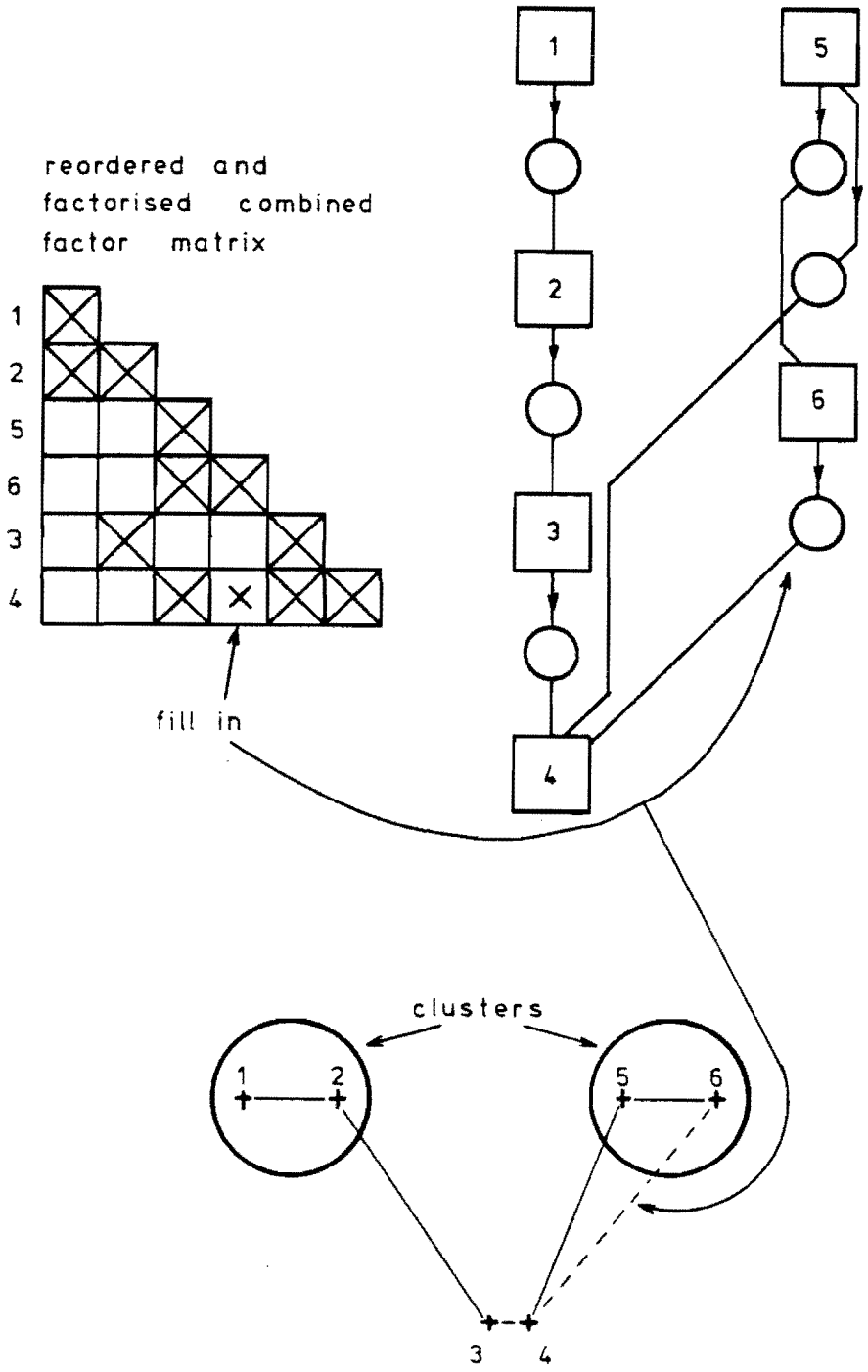


Figure 9.11: Reordering to Increase Task Independence in Network 2

9.6.1 Search Length

According to the PBIF algorithm, searching for ready tasks is achieved through examination of a vector whose elements correspond to the readiness of each column related task. As execution progresses all tasks up to a certain column will be initiated. The most likely area in which new ready tasks will be found is in those related to subsequent nearby columns. Therefore, to save time, searches could be restricted to a fixed range in which it is likely that most ready tasks would lie. The trade off in determination of the optimal search length is between the time wasted searching where ready tasks are unlikely to exist, and the loss of scope for exploitation of parallelism implied by neglection of possible ready tasks.

The ratio of time to complete a single search sequence to that required in execution of a typical column related task is very relevant in selection of search length. For large processors the ratio will be high, due to the closeness of floating point and other instruction execution times, so time spent in searches is correspondingly significant. On the other hand, for small processors, such as the 8086, the ratio is low so the optimal search length will be biased towards a large value, and the effect on performance resulting from changes in search times will be small.

Practical investigation was impaired by these problems. Under any conditions, execution times change slightly from run to run. Many studies were made and, to varying degrees, in all cases difficulty arose in isolating the effect of search length variation. From all cases investigated, the one in which variations due to changes in search length are most easily viewed is presented. The configuration involves five processor operation using the 35 bus network. Figure 9.12 illustrates the measured performance variation. The estimates of error applied to each

measurement are based on small samples and are, therefore, very approximate. Further accuracy would be difficult to usefully apply for reasons outlined in section 9.8.1.

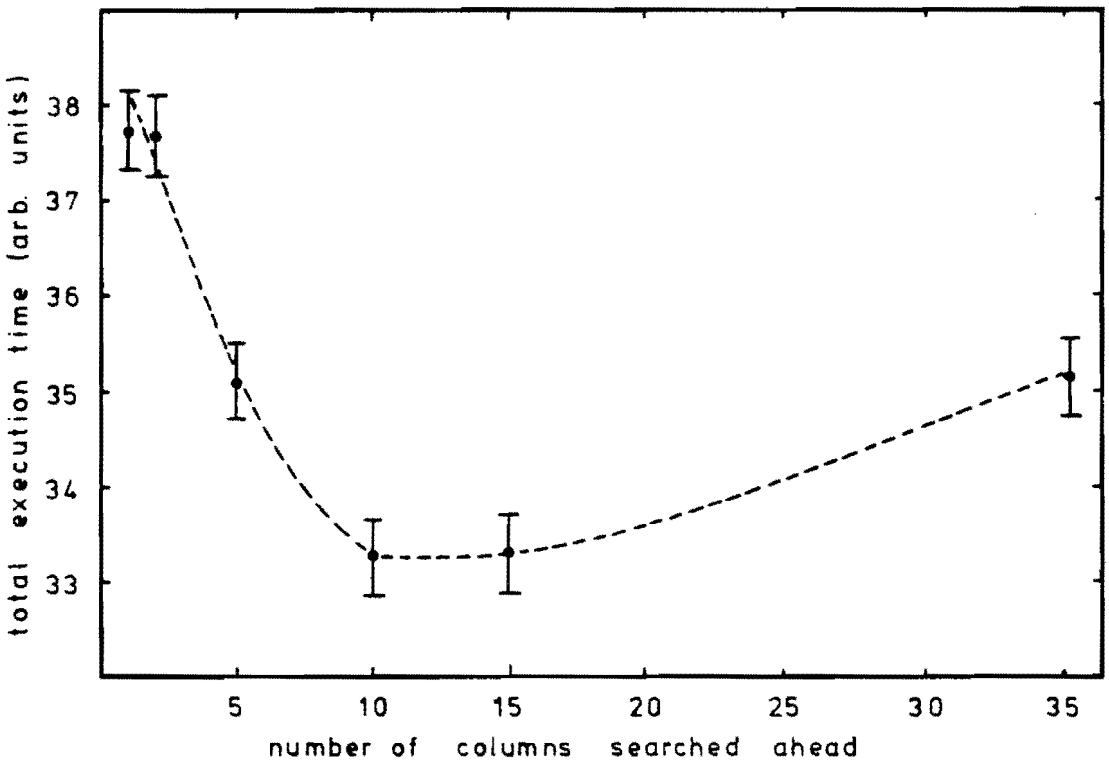


Figure 9.12: Effect of Varying the Search Length

Optimal search length appears to occur between values of 10 and 20. As is clear in the figure, the confidance which could be applied in determination of this length is limited. For a variety of networks, the optimal search length could vary greatly as a result of differing distribution of elements. Similar investigations, using BIFSIM, could provide performance information for processors such as the B6700 in which the effect would be more distinguishable. However, the accuracy expected of BIFSIM in this application cannot be ascertained as no reliable real system comparison is available.

Optimal search length will depend on the number of processors and their complexity; and network size, sparsity, and distribution. Consideration of an appropriate search length becomes more important for more complex processors. No conclusive determination of optimal length has been possible with the 8086. However, the effect has been demonstrated. In practice, having a known configuration, the optimal search length could be established through trial and error, and subsequently fixed at that level.

9.6.2 Semaphore Examination

The level of bus conflict can be estimated using information such as the number of processors, task execution times, and the number of bus accesses required by each task. For example, the adequacy of MULTIBUS for the UCMP system was established using this type of information (see section 6.2.2.2). However, accurate determination of the number of bus accesses is difficult. For instance, any number of bus cycles can occur in a single semaphore setting operation. Many of the accesses made are unfruitful, and therefore unnecessary. If many processors simultaneously attempt ssemaphore setting operations, bus conflict and related idling overheads could become significant. Therefore, it is proposed that performance could be improved through introduction of delays in setting routines. Bus conflict can be traded off against the possible loss of useful processing, caused by delaying processors, to select the optimal delay period.

In practice, the UCMP system, with its limited number of relatively slow processors, rarely encounters bus conflict. Even using the minimal possible delay in semaphore setting, hardware related idling overheads are insignificant. All that is observed, therefore, is a degradation in performance as the delay is increased. For systems with many more

processors, though, the bus conflict problem could form a serious constraint so selection of an appropriate delay could be valuable. Like the choice of search length, the optimal delay could be determined and set for a particular implementation.

9.7 Bus Conflicts

In the development of highly parallel computers, the possibility of bus conflict provides impetus for the inclusion of complex processor-memory interconnection networks. The UCMP system bus needs were specified using the expected conflict levels implied by the PBIF algorithm. Overall performance degradation due to the problem was expected to be in the order of 0.1% (see section 6.2.2.2). In addition, conflict was expected to result in bus saturation for approximately 100 processor operation.

The UCMP system was applied to an investigation of bus conflict problems. Such a study has been undertaken with three intentions:

.to confirm that the approach to specification of the UCMP system bus was valid, and determine what margins for error really existed,

.to find the point at which bus saturation is reached and how it was approached, and

.to enable consideration of reliable methods of bus conflict modelling when simulating parallel execution.

9.7.1 Bandwidth and Conflict

L.S.Haynes et al (1982) suggest that for a ring structure (ie. a single interconnecting bus) with n processors 'only 1/n of the bus bandwidth is available for each of the n processors'. It is concluded,

therefore, that such a network 'is suitable only where communication requirements are very small...'. The implication in this reasoning, that a bus with capacity for n times the requirements of a single processor is necessary, is, in fact, optimistic. 'Bandwidth' is a commonly used, but loosely based, term when referring to bus capability. Bandwidth is not as easily distributed as, for instance, radio channels.

A single processor uses a bus uniformly in time. 100% bus utilisition, ie. full use of bandwidth, can be reached without introduction of overhead. Hence, processors are designed with well matched bus interfacing and instruction execution times so that buses are no more complex than necessary, and a high proportion of bus bandwidth is exploited.

Non-uniform bus utilisation, as implied by randomly occurring requests for access to global memory in a multiprocessor, results in overhead well before the full bandwidth is used. If a bus is busy only 10% of the time then, for a particular processor, the probability that an access will be delayed is about 10%. As the use of bandwidth increases towards 100%, overheads quickly become unacceptable. Detailed statistical calculation of overheads is plagued by a lack of accurate information of the true distribution of bus requests. A reasonable approximation for small overheads, which is the range of interest, is : For a bus which is busy X% of the time, the increase in average access time will be X%. Therefore, rather than exploiting 100% of available bandwidth as is practical for single processors, 5% to 10% is probably more appropriate for shared buses.

9.7.2 Modelling Bus Conflict

Three methods could be applied to induce high levels of bus conflict:

.the algorithm implemented could be changed to involve a heavier requirement for access to MULTIBUS, or

.the number of processors could be increased, or

.the presence of additional processors could be modelled.

As no change from the PBIF algorithm is desired, and it is not feasible to introduce further processors, the third approach is the only realistic option.

9.7.3 Processor Modelling Techniques

Three approaches aimed at emulating the presence of processors were considered:

(a) the response time of globally accessible memory could be artificially extended.

(b) the clock which synchronises operation of MULTIBUS could be slowed down. Priority resolution, which forms a part of every bus cycle, would then take longer.

(c) special hardware, with no processing capability, could request use of MULTIBUS in the same way that real processors do.

No information is available describing the time distribution of bus accesses which can be expected from a processor or how this distribution varies as bus utilisation increases. Therefore, assumptions must be made and qualitative judgements applied in creation of processor models. Also,

the limitations of the any models selected will be difficult to evaluate.

The UCMP hardware imposes a constraint, called the 'minimum bus availability period', on modelling techniques. Whatever changes are made to the availability of MULTIBUS, all real processors must be guaranteed access at some stage. To achieve this, the periods during which MULTIBUS is made available must exceed a fixed length ie. the time taken for all real processors to access MULTIBUS once. If this requirement is not met, one or more processors could be continuously denied access to global memory with consequent probable operation locking (see section 9.8.2).

Neither method (a) nor (b) allow flexibility in the time distribution of accesses made by modelled processors. Using method (b) processors can be denied MULTIBUS access when the clock period exceeds a limit. Both methods (a) and (c) imply non-trivial hardware changes. For method (a) the changes imply undesirable adjustment to commercially supplied equipment. Therefore, because it offers modelling flexibility and has acceptable hardware requirements, method (c) was selected and implemented.

9.7.4 Method Implemented to Induce Bus Conflict

During any access via MULTIBUS, the first bus clock cycle is used to select, from among the currently requesting bus masters, the one with the highest priority as defined by the priority resolution hardware. If the bus master with the highest possible priority maintains a continuous request then all other processors will be inhibited from using MULTIBUS. Hardware has therefore been developed allowing manipulation of the highest priority request line and, thus, control the availability of MULTIBUS.

Ideally, the pattern of bus availability would be identical to that existing in the presence of additional processors. Because of the

constraints implied by the UCMP structure, the true distribution cannot necessarily be implemented even if it could be defined. Two realizable distributions were considered:

.a constant frequency signal giving regular use of MULTIBUS to real processors, and

.a signal with pseudo-random period providing random availability of MULTIBUS.

Unrealistic operation, when using a pseudo-random signal generator, led to the choice of a constant frequency approach as the only useful option. The unrealistic nature of operation results from the minimum bus availability period constraint implied by the UCMP hardware. To guarantee sufficiently long periods of bus availability, the maximum period during which the bus can be unavailable quickly becomes excessive as the bus availability is reduced.

A signal generator enabling constant mark space ratios with variable frequency was used to provide insight into the effect of changes to distribution. Consider first the highest possible frequency if, for instance, the bus is available 10% of the time. At very high frequency a typical processor would find access time extended by a factor of 10. However, as such accesses represent only a small proportion of the execution time, overall performance degradation is low. Alternatively, when the frequency is very low, the bus becomes available for long periods, during which unimpeded operation occurs. However, the whole system is effectively switched off for the remaining 90% of the time. Therefore, over a wide range of frequency, the system performance would be expected to vary from nearly unaffected to 10 times performance degradation. Using the UCMP hardware the true variation was plotted, figure 9.13, for a single

processor to a point where performance degradation was significant.

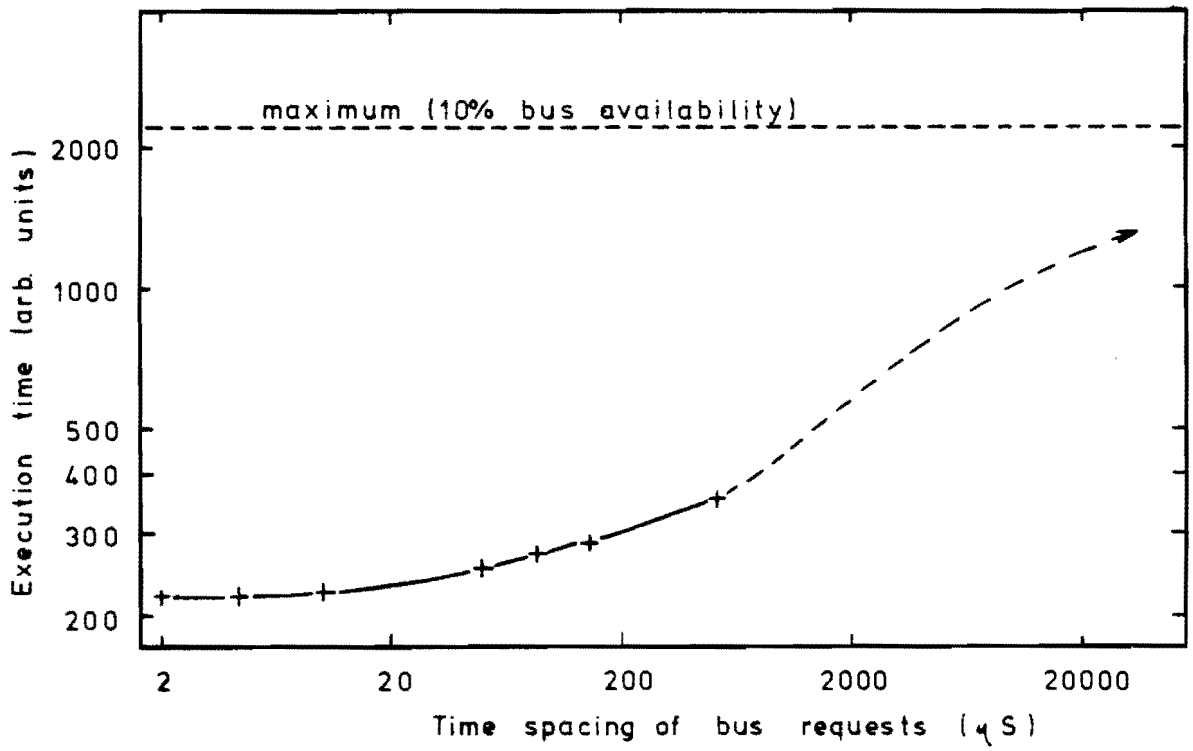


Figure 9.13: Performance Variation Due to Changes in the Frequency of Bus Availability with Constant Average Bus Availability

As any distribution, whether regular or random, with a given mark space ratio must result in performance between the limits expected when using a regular pattern with varying frequency, it can be concluded that regular patterns can be used to accurately model the presence of additional processors. However, it is not possible to determine the frequency at which a good model of additional processors is provided. For the investigation which follows, the maximum possible frequency was employed. As the bus attenuation factor increases the frequency is correspondingly reduced, as would intuitively be expected from real system operation. The choice of frequency is, though, an assumption which should be remembered in interpreting the results which follow.

9.7.5 Results

The extent of the overhead due to bus conflict, as it changes with the number of processors, could be described in various ways. An expression is required which sums up the effect in an easily interpretable manner with independence from other overhead sources. Such a measure is achieved through use of a term defined here as the bus saturation factor (BSF).

The factor indicates the effectiveness of the addition of further processors. Its value corresponds to the proportion of ideal gain which is lost due to bus conflict. When very low, therefore, bus conflict is insignificant, and when equal to unity no further gain in performance can be achieved by using any number of additional processors. The value can even exceed unity if additional processors impede the progress of those performing useful processing.

For the performance measurements made, bus saturation is estimated using a difference equation as follows:

$$BSF = \frac{\text{(decrease in performance / performance)}}{\text{(increase in bus availability attenuation / bus availability attenuation)}}$$

$$= \frac{\text{(increase in execution time / execution time)}}{\text{(increase in bus availability attenuation / bus availability attenuation)}}$$

$$= \frac{(\Delta te / te)}{(\Delta Baa / Baa)}$$

$$= \frac{Baa}{te} \frac{(\Delta te)}{(\Delta Baa)}$$

where: te - execution time
Baa - bus availability attenuation

The number of processors modelled (Nm) can be expressed as the product of the bus availability attenuation and the number of real

processors (Nr).

$$Nm = Baa\ Nr$$

Bus saturation can, therefore, be determined as a function of the bus availability attenuation for each number of real processors. This is the approach used in measuring the performance depicted in figure 9.14.

Points to note from the three sets of characteristics are:

.The absolute and relative execution time curves are initially flat and a change of slope occurs where conflict becomes significant.

.The change of slope occurs at around 100 modelled processor operation, but, at this stage, performance is already severely degraded.

.The degree to which saturation is taking effect is most clearly portrayed in the bus saturation factor plots.

.Acceptable levels of bus saturation occur only at very low numbers of modelled processors, perhaps around 10 to 20. This can be seen in the plot of bus saturation factor as the point at which it begins to become observable.

A check which supports the validity of the processor modelling approach can be implemented by comparing one and two real processor operation. For a fixed number of modelled processors, the bus saturation factor should be unaffected by the number of real processors. In terms of bus availability attenuation (used in the plots), a factor of two difference should arise between the characteristics related to one and two

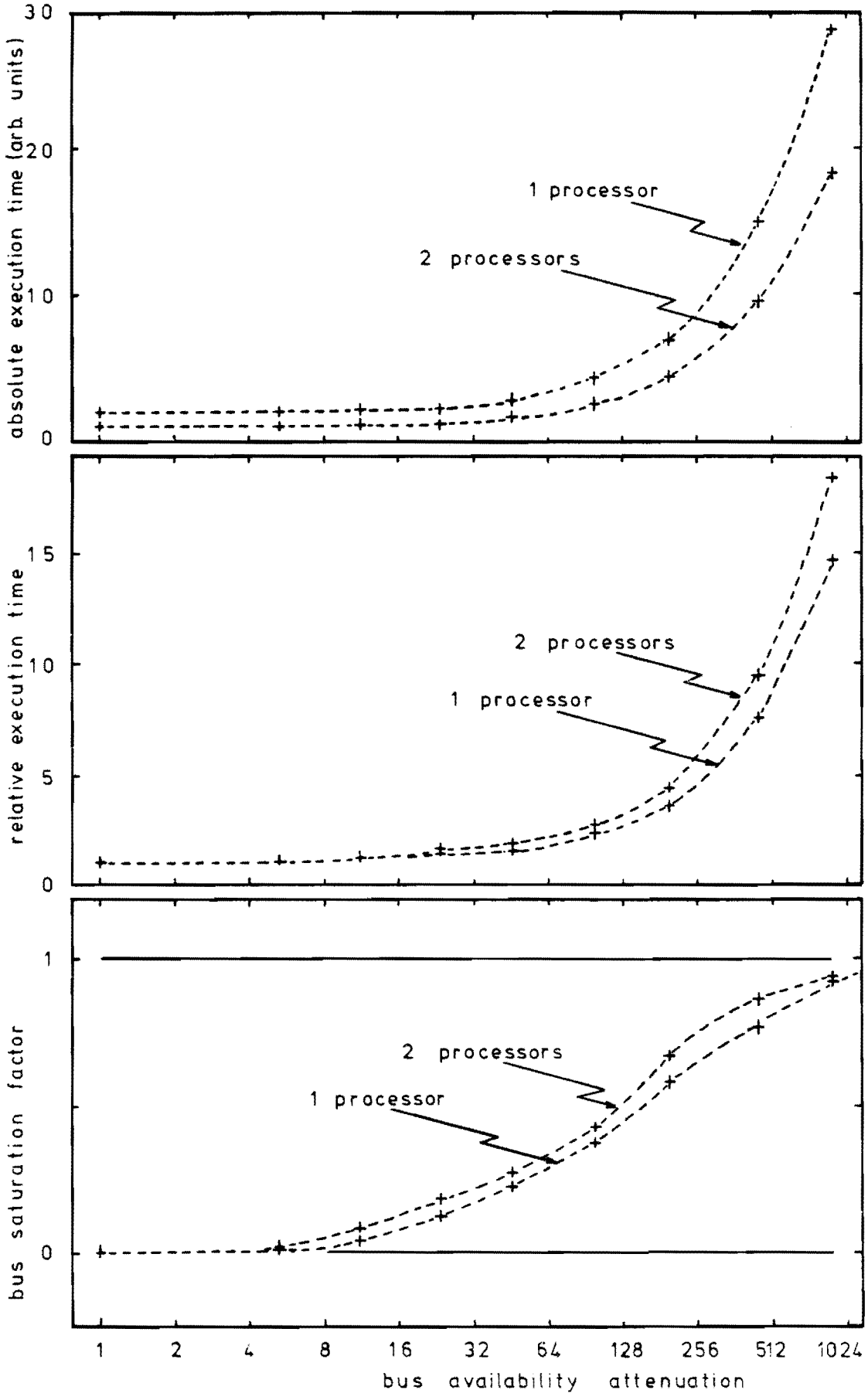


Figure 9.14: Performance Variation due to Changes in Bus Availability

processors. By observation, it can be seen that the horizontal separation between the bus saturation factor (and relative execution time) characteristics is close to a factor of two.

9.8 Experience with the UCMP Hardware

Performance measurements are affected by hardware initiated variations in execution time. The problem is not sufficient to interfere with overall performance evaluation, but does create problems eg. during parameter tuning.

Another problem, which fortunately does not affect performance other than during tests of bus conflict, arises as a result of the MULTIBUS priority resolution scheme used.

9.8.1 Non-Constant Execution Times

For any fixed number of processors, measurements of execution time vary from run to run. Note that the figures presented throughout this and the next chapter are selected from groups of measurements. The level of differences occurring can be separated into two categories:

.small, apparently random deviations of less than 1% of execution time, and

.larger differences, generally around 5% of execution time. Measurements tend to fall consistently into groups separated by this difference.

The feature identified as the source of both these levels of deviation is the set of assynchronous clock signals used in the UCMP hardware. Each processing element has an independent clock, and each accesses MULTIBUS which has a further arbitration synchronising clock.

The number of wait states, and the order of access to MULTIBUS can depend on the phasing of clocks at points during execution. The slight changes in execution time this causes results in the small, random deviations.

These minor changes in execution time can occur at crucial points eg. changing the order of completion of tasks. Such changes can result in subsequent alteration of task distribution and execution sequence, possibly resulting in significant changes in total execution time. This situation accounts for the second, larger execution time deviation category.

9.8.2 Priority Resolution Problems

In the design of the UCMP system, capability for many bus masters was required, and no need was seen for all to have exactly equal priority for use of MULTIBUS. For these reasons, the fixed priority scheme was implemented. This selection introduces a possible execution time fault situation. The possibility of its occurrence emerged during the design of processor models for bus conflict studies.

The potential problem can be described in terms of an example: A low priority processor sets a semaphore. A number of higher priority processors then attempt to set the same semaphore. As such, each of them accesses MULTIBUS frequently. In combination, they may use MULTIBUS continuously. If so, the low priority processor will not be able to release the semaphore, and operation will become locked.

Fortunately, because the minimum period between bus accesses during semaphore checks is sufficiently long, the problem does not arise in the UCMP system. The possibility, however, is an important consideration which may provide incentive for the use of, for example, rotating priority arbitration.

9.9 Conclusions

A strong similarity was found between performance simulated using BIFSIM, and real operation of the UCMP system. As a result, confidence in the accuracy of the simulator is increased substantially.

A variety of real power system data sets were found to introduce, to differing extents, an ill-conditioning problem. The problem originates in the form of networks and is worsened by the 'serial optimal' ordering scheme used. A return to alternative ordering schemes, such as block related methods, is shown to be a possible approach to performance improvement. The choice of an optimal ordering scheme is a complex problem because, for instance, it is dependent on the number of processors.

Insensitivity to changes of program parameters in the UCMP implementation made optimisation difficult. The importance of appropriate choice of these parameters in different hardware environments, where they could have a marked effect, is discussed.

A hardware model of additional processors was used to examine the effect of bus conflict on performance. To achieve significant bus conflict, a high level of extrapolation was necessary. Because of this and difficulties in determining the extent to which the model emulates real processors, confidence in results is limited. However, the design expectation that saturation would occur at approximately 100 processor operation in execution of the PBIF algorithm was illustrated.

CHAPTER 10

HARDWARE PERFORMANCE - INCLUSION OF GENERATOR MODELS

10.1 Introduction

The physically independent siting of non-network components, such as generators, within power systems is reflected in their models during the execution of simulation programs. Therefore, higher levels of effective processor utilisation are expected when comparing performance with that occurring during the heavily dependent solution of linear equations.

Non-network component models provide a representation of all power system elements not described in the linear network equations. Axis conversions between, for example, generator and network frames of reference are included in associated non-network component models.

Network and non-network related serial execution times are similar. Hence, the expected improvements in performance due to the independence of non-network models will be constrained by the linear substitution processes.

Beyond performance improvements due to the independence of non-network models, an opportunity for further gains arises due to parallelism between network and non-network models. Even assuming maintenance of the linear solution scheduling scheme described in Chapter 4, there are a wide range of scheduling possibilities when non-network models are included.

To date, very little has been published describing methods for efficient inclusion of non-network models. Examples of what has been

written include early work, such as that by I. Durham et al (1979) which did not consider the possible overlap of network and non-network tasks. More recently, some mention has been made of the use of such overlapping, eg. (Brasch et al, 1981). In this case no attempt at optimisation is described. Currently, therefore, there is much scope for development and improvement aimed at determination of the most efficient approach.

Three methods of non-network related task scheduling are described in this chapter. Tests are made to ascertain their expected performance. Two power system descriptions are used as benchmarks. Of these the most appropriate is identified and employed in extending tests to execution of a full time step.

10.2 Constrained Ideal Performance

To assess the value of algorithms for the simultaneous solution of network and non-network models it it useful to have some idea of the likely limits to execution performance. The known linear solution characteristics ensure that the unconstrained ideal performance curve is an impossible objective. A realistic performance target takes account of the known inefficiencies in linear solutions.

The constrained ideal performance characteristic is defined and generated by the equation:

$$\eta(i) = (\alpha(1)+\beta)/((\beta/i)+\alpha(i))$$

where: η - the constrained ideal speed up

i - the number of processors

$\alpha(i)$ - the time taken in execution of the linear solutions using 'i' processors

β - the time taken to serially execute the non-network models

The performance predicted by this equation would occur if:

.the non-network related tasks were distributed and executed with no execution time wastage by any processor, and

.network and non-network related tasks were serially separated.

As such, the curve is not truly ideal as advantage may be taken of the parallelism between network and non-network related tasks.

10.3 Execution Sequencing

The UCTS program simulates the state of a power system at each of a number of regularly spaced instants. The period between instants is called a time step. Information produced at one time step provides the initial conditions for the subsequent step. Consequently, parallelism between time steps is limited. F.L. Alvarado (1979) describes a method utilising parallelism between time steps. Dependence between time steps, however, results in extensive matrix fill-in when attempts are made to create parallel executable blocks. Although it is not clear that the fill-in problem is insurmountable, the approach has not been developed further. This is probably because far more scope for exploitation of parallelism exists within time steps. Hence, consideration in this chapter is restricted to this case.

Generators are the only non-network components modelled in the execution trials to be described. When considering dependence and execution sequence, other elements, such as D.C. converters and induction motors, have models similar to those of generators ie. they affect single nodes in the network model.

10.3.1 Single Time Step

Within each time step, a solution results from an iterative sequence which, in terms of physical components, involves information transfers between non-network and network models until convergence. The order of execution is shown in figure 10.1. Initially, non-network component models are integrated, and associated non-integrable variables are linearly extrapolated. Values required in implementation of trapezoidal integration, and which remain constant throughout this time step, are also evaluated. This initialising step is followed by iterations between network and non-network models until convergence criteria are satisfied.

Between the forward and backward substitution phases is a point at which synchronisation of all processors can be assumed. (This assumption was made in the PBIF algorithm, and is based on the very limited scope for exploitation of parallelism between these groups of tasks.) Taking account of the dependence between (1) time steps, and (2) the forward and backward substitution steps, three serially separable execution sequences are identified as:

.GF - Initial non-network integration and the forward substitution of network equations.

.BGF - Backward linear substitutions, subsequent integration steps, and forward substitutions.

.BG - Backward substitutions and a final integration step.

These steps can be combined to form complete time steps as illustrated in figure 10.1. In execution time based performance measures, knowledge of the individual speeds of these three sections can be combined to evaluate expected performance for various numbers of iterations.

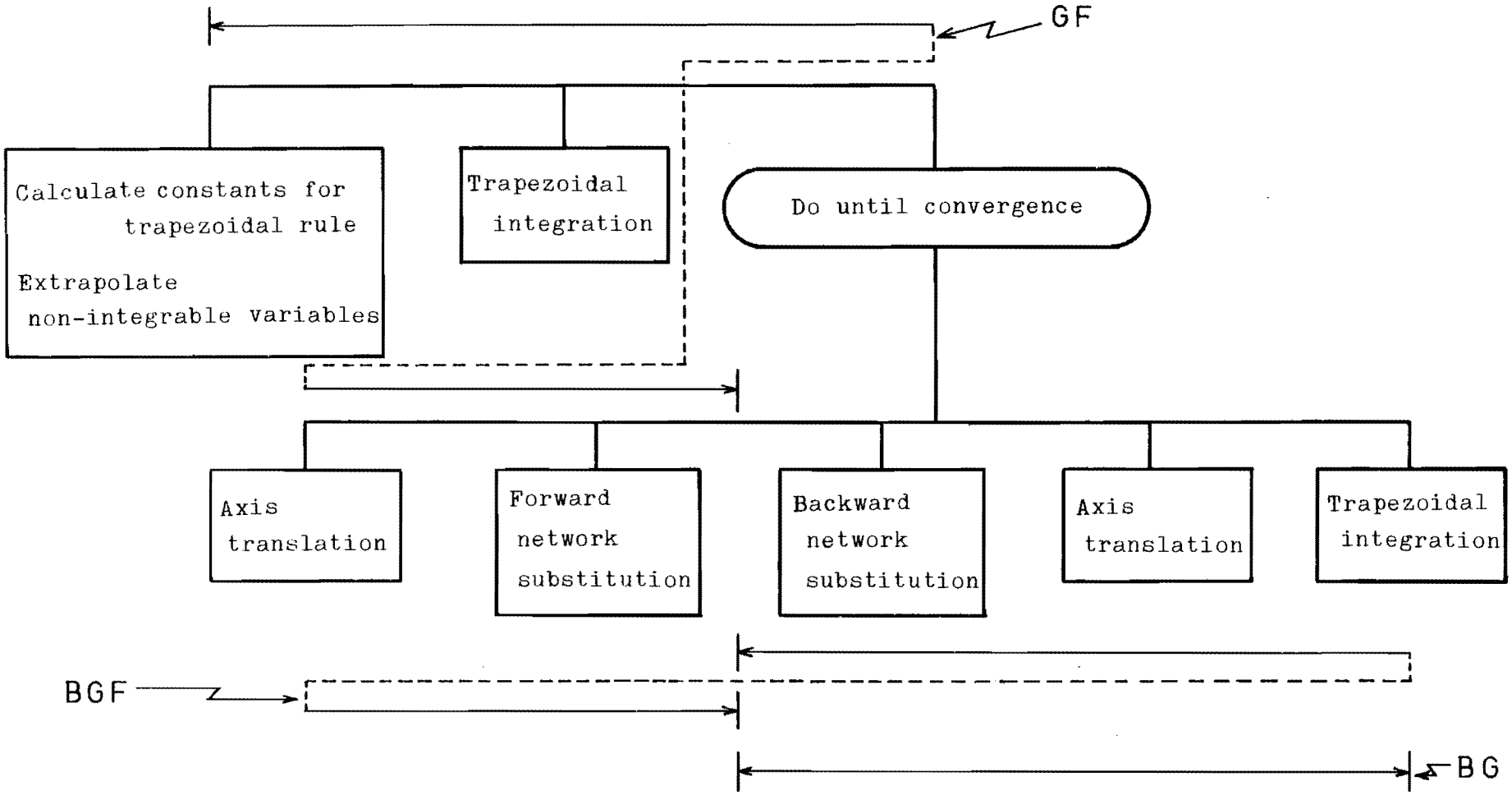


Figure 10.1: Program Sections for a Single Time Step

10.3.2 Predominant Loop

The task graph in figure 10.2 illustrates the BGF execution possibilities. The GF and BG sequences can be represented by similar graphs in which the backward and forward substitution tasks respectively are disregarded. For simplicity, the graph does not allow non-network elements any association with more than one network node. However, devices such as AVR's, which can introduce a second node, can be easily modelled. The task graph can be viewed as an extension of figure 4.1 with the basckward substitutions included, and with the following conditions being placed on task initiations:

.non-network models can be executed once information from associated nodes in backward substitution is available, and

.forward substitution tasks can commence once preceding network requirements are fulfilled (ie. according to the linear solution algorithm) and, if present, completion of execution of the non-network component model at that node.

10.4 Benchmark System Data Selection

When practically implemented, the number of processing elements in a multiprocessor would probably be at least an order of magnitude fewer than the number of nodes in simulated power systems. As such, reasonable utilisation of processing resources is obtained. To observe the limitations of parallel processors, however, smaller system models are needed. Selection of an appropriate system here is based on the results of linear substitution investigations which were documented in Chapter 9.

Two system data sets were considered to be of roughly suitable order: a 24 bus, 5 generator system, and a 35 bus, 14 generator system.

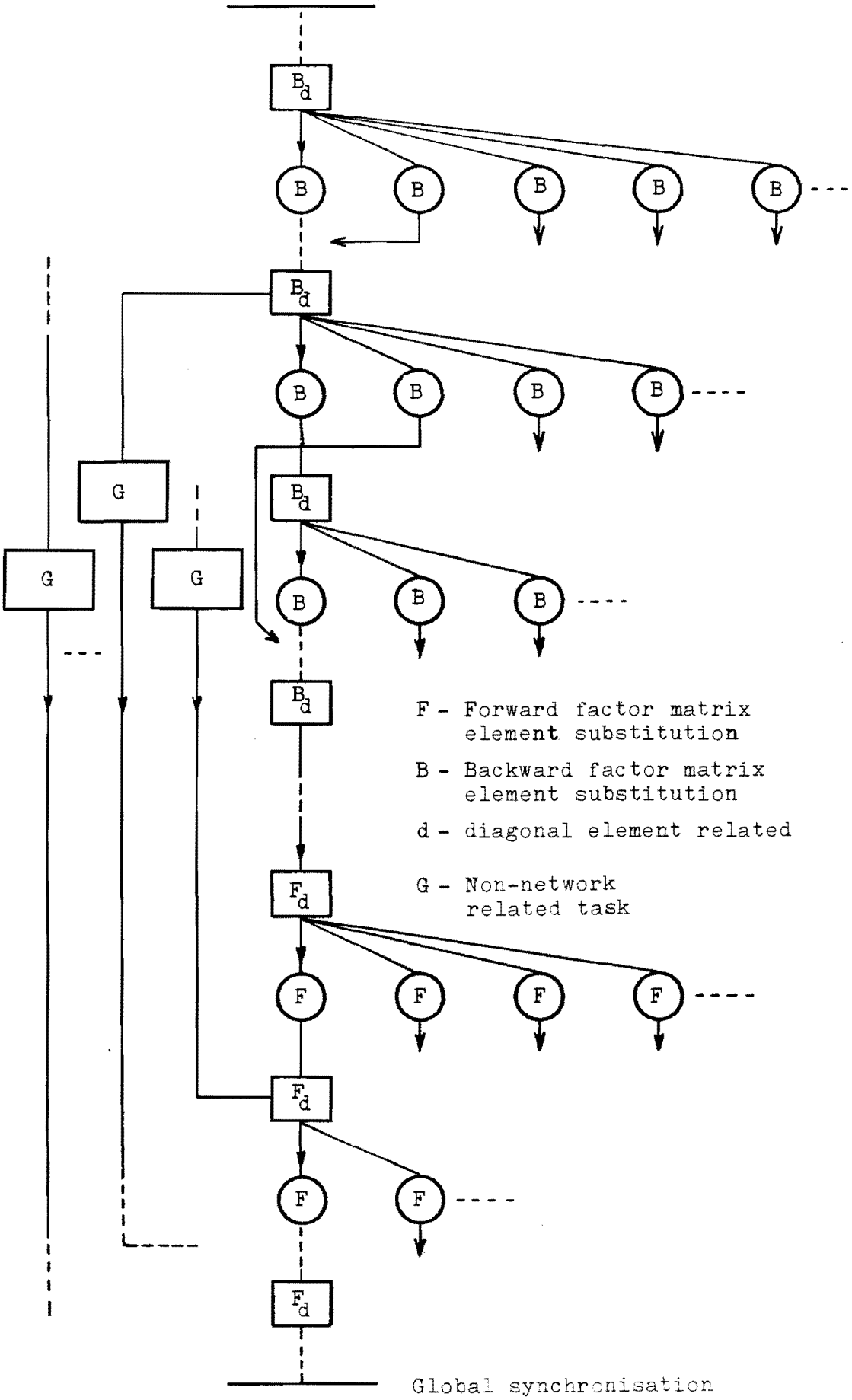


Figure 10.2: Independence of Network and Non-Network Related Tasks

The linear solution and constrained ideal performance characteristics of both systems are presented in figures 10.3 and 10.4. Note the effect of the higher proportion of serial processing time being devoted to non-network solutions for the 35 bus system (at 70% as opposed to 46% for the 24 bus system) ie. for the larger system the constrained ideal curve is lifted further towards the unconstrained ideal from the linear solution performance characteristic.

The 24 bus system is considered to represent the most valuable benchmark because: it has a typical distribution of processing between sections, and its expected performance characteristics approach saturation with the number of processors available. However, performance measured using the 35 bus system is used to compare and substantiate those achieved with the 24 bus system.

The number of non-network components in the 24 bus system is fewer than the maximum number of processors. Therefore, any effects resulting from the close relationship between the number of available tasks and the number of processors may be observed.

10.5 Non-network Task Scheduling

The scope for wide distribution of tasks among processing elements is shown (figure 10.2) to be enhanced by the inclusion of non-network component models ie. network and non-network related execution sequences are independent in many cases. Scheduling algorithms capitalising on these opportunities can take a wide variety of forms. The approaches described here are not intended to explore every available avenue in search of an optimal technique. Instead, the methods are introductory and illustrate the possibilities which exist.

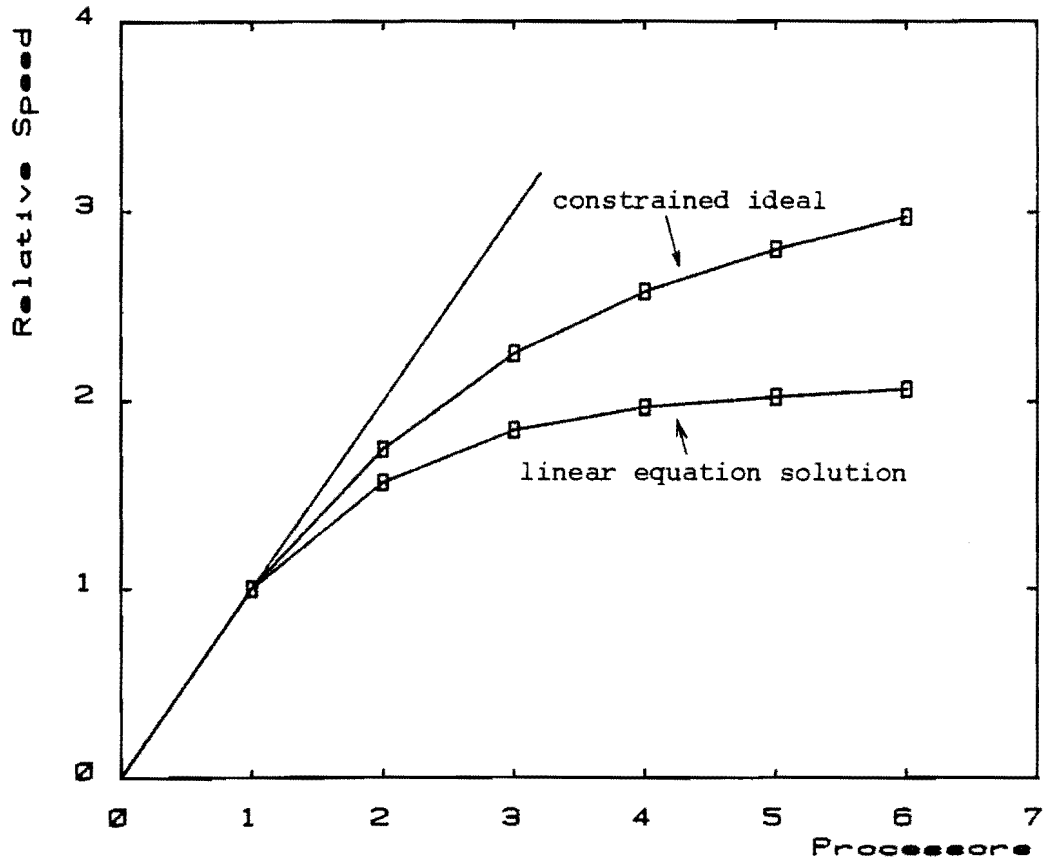


Figure 10.3: Ideal Performance When Constrained by Linear Equation Solution Steps - 24 Bus System

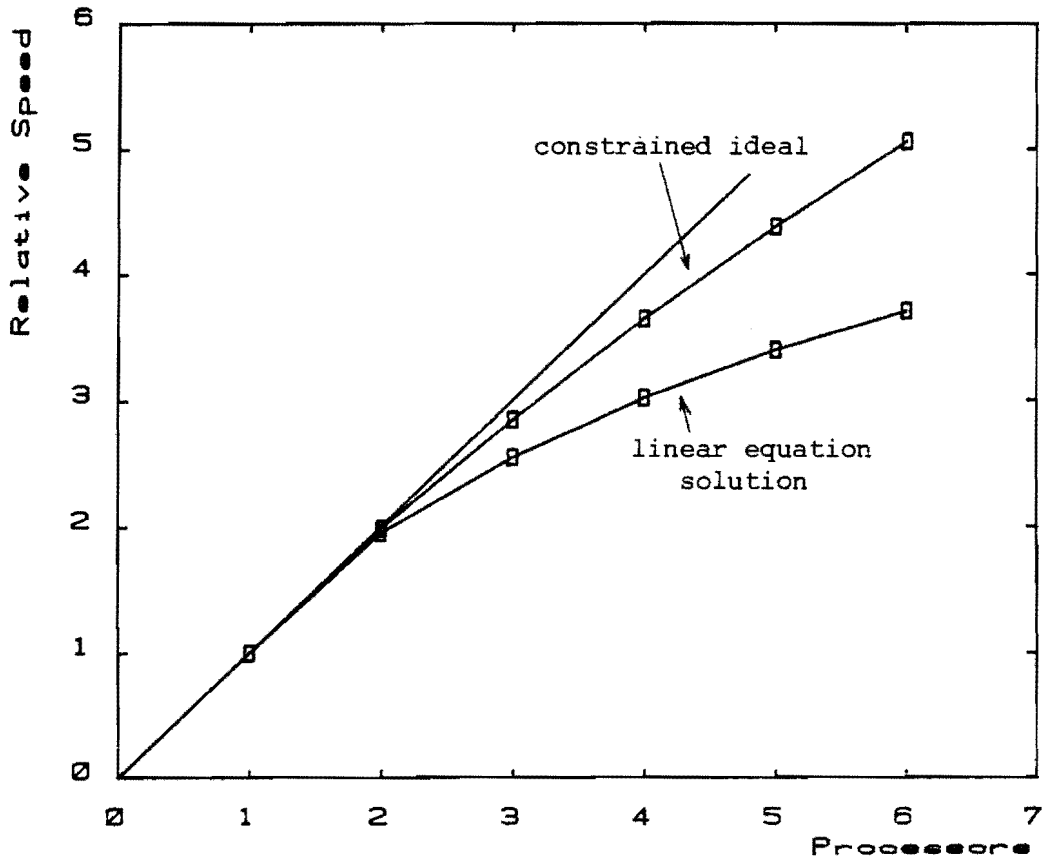


Figure 10.4: Ideal Performance When Constrained by Linear Equation Solution Steps - 35 Bus System

Three scheduling algorithms, referred to as serial, network priority, and non-network priority schemes, have been specified and implemented. The latter two attempt to utilise the natural structure of the problem. Dependence on the structure of networks makes a choice between them difficult without consideration of practical performance.

10.5.1 Serial

Serial scheduling involves separation in time of network and non-network solutions. Hence, no advantage is taken of the independence of network and non-network related tasks. The strict change in execution sequence which this implies is illustrated in figure 10.5 which is a special case of the task graph depicted in figure 10.2. Non-network task management is easily implemented as entry requirements and task completion identification are far simpler than those during linear substitutions. Global synchronisation at points between the network and non-network steps is also easily arranged.

10.5.2 Network and Non-network Priority

During backward substitution occasions arise at which further substitutions or a non-network component model could be initiated. These points occur where nodal information which satisfies the input needs of a non-network model becomes available. If network and non-network related executions are to be combined, tasks can be selected from either category. The algorithms developed differ in the category searched first for ready tasks.

Qualitative consideration of likely means of efficiency improvement leads to a conflict between the possibilities:

.Non-network component model execution reduces the number of

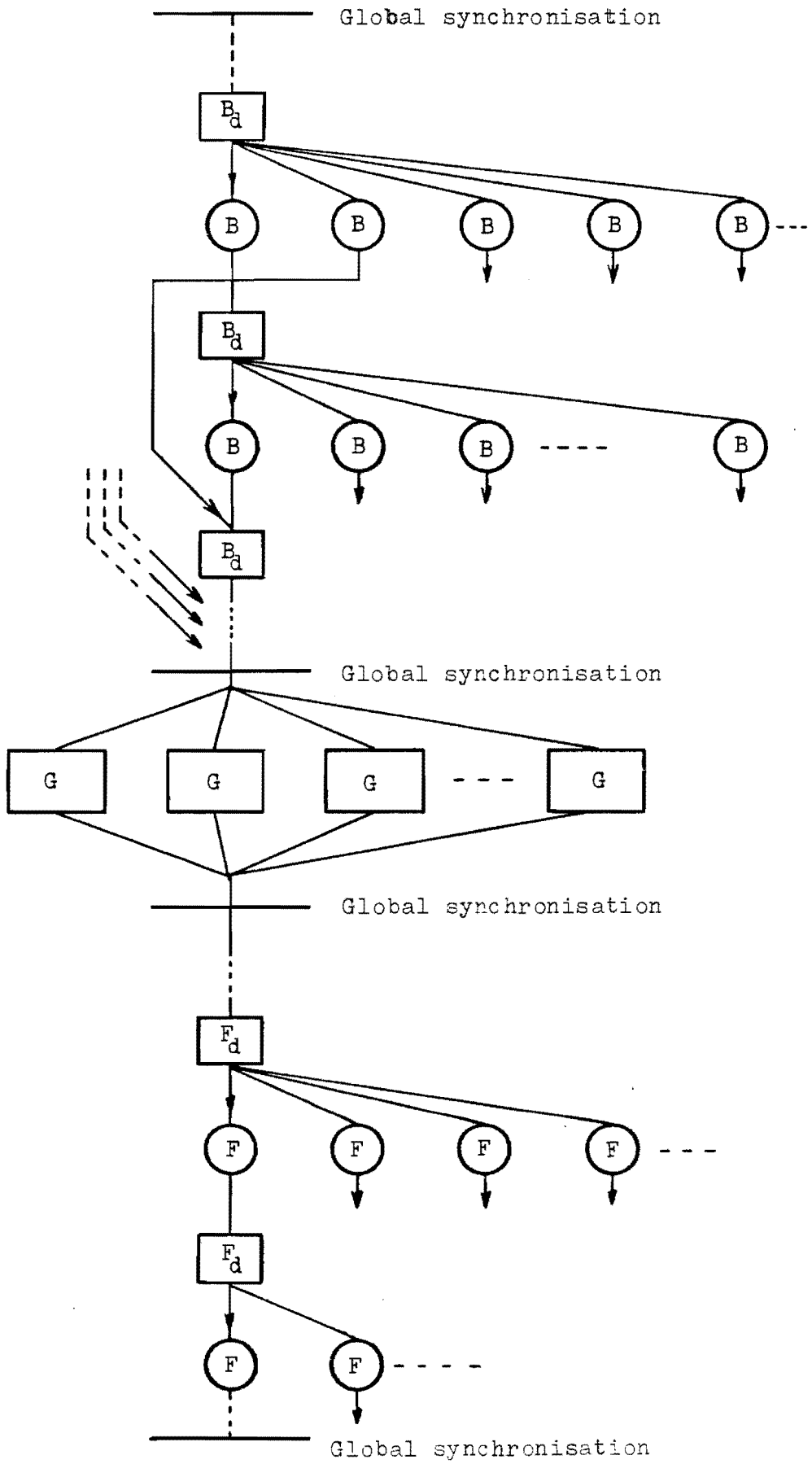


Figure 10.5: Serial Approach to Inclusion of Non-Network Related Tasks

processors involved in substitution. Hence, efficiency is increased. The greatest increase occurs if non-network components are modelled at the point where substitution processes are least efficient ie. towards the end. This argument leads to the conclusion that generator models should be given low priority resulting in their execution as late as possible during backward substitutions.

.During forward substitutions some tasks are dependent on information provided by non-network models. Hence, execution can be interrupted by late completion of execution of these models. In fact, it is likely that the last non-network task initiated will be the first required to be complete. Consequently, during forward substitutions it would be preferable for non-network routines to have had higher priority resulting in their earlier completion.

Although this argument is far from rigorous it is clear that neither approach stands out as being preferable. Practical tests of performance with real systems are therefore desirable, not only to establish the value of the methods, but also to compare them.

10.6 Programming Considerations

Non-network models employed originated in the UCTS program. Conversion to allow parallel execution was simplified by the original execution sequence in which non-network component states are evaluated sequentially. This allows division into individual component related tasks with very minor changes to source code.

Serial scheduling of non-network tasks requires simple management. A vector is used to record the state of each task. Initially all tasks are ready. As each is selected status is changed to running. Detection of

completion is achieved through observation of a counter which is initialised to the number of tasks. As each task is completed, the counter is decremented.

More complex management requirements arise in implementing the network and non-network priority schemes. The requirements for both schemes, though, are similar. Compared with serial scheduling, two new areas emerge at which task readiness must be determined. The first (a) is in the initiation of non-network tasks during backward substitutions, and the second (b) is the set of checks needed to ensure that dependent non-network tasks are completed during forward substitutions. The approaches adopted in meeting these requirements are as follows.

(a) Non-network tasks are ready once information becomes available from associated nodes during backward substitution. Detection of readiness is achieved through observation of the number of non-substituted elements in each backward row (see Chapter 4). (Note that the absence of non-singular diagonal elements in backward substitution factor matrices is necessary to ensure the integrity of this approach.)

(b) Task readiness in forward substitutions is detected through observation of the number of non-substituted elements in each forward row. Hence, as non-network related tasks impose a further dependent input requirement on associated substitutions tasks, management is achieved through increments to elements of the 'NUMBERINROW$FOR' vector (see Chapter 4). As each non-network task is completed the associated element is decremented. Consequently, searches for forward substitution tasks do not change from those used for linear substitutions only.

A listing of program sections forming the basis of the network priority scheme is given in Appendix 5. The BGF related search and coordinating routines presented illustrate the role of a slave processor.

10.7 Execution Performance

Performance characteristics are presented for each of the three scheduling algorithms. All are based on practical measurements using the UCMP system. For the serial approach, results for both the 24 and 35 bus systems are given. For the other schemes characteristics based on the 24 bus system only are presented.

For each scheduling scheme performance for each of the three serially separable code sections is illustrated. Finally, for the 24 bus system, this information is combined to form performance characteristics for complete execution of a time step.

10.7.1 Serial Scheduling

Figures 10.6 to 10.8 illustrate the performance measured for each code section using the 24 bus system data. Comparing these figures with figure 10.3 shows that a significant improvement in performance is achieved when compared with linear solutions only.

Similar data is presented in figures 10.9 to 10.11 for the 35 bus test system. Because the proportion of execution time required in solution of non-network models is higher, the gains in performance are also higher as was predicted in figure 10.4.

An interesting feature, common to all the 24 bus based characteristics, is the leap in performance between four and five processor operation. This is a result of the number of non-network components modelled ie. five generators. The reduction in performance up to four

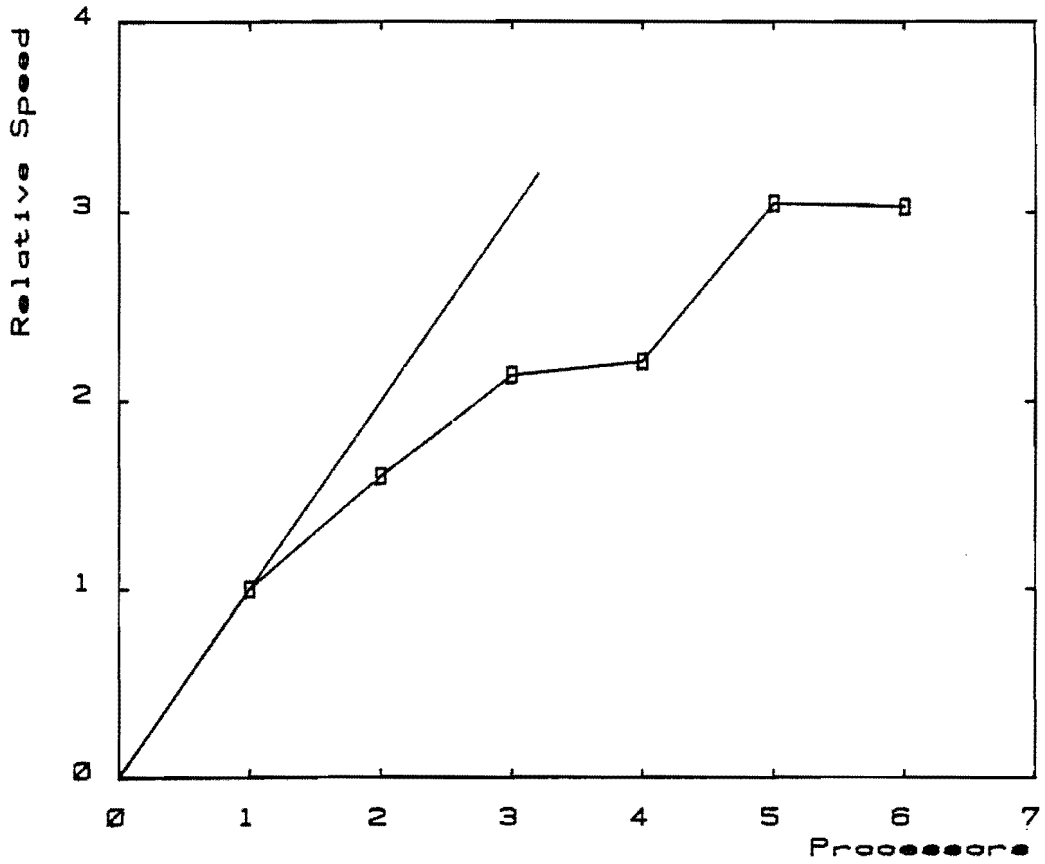


Figure 10.6: Performance for GF Program Section - 24 Bus System

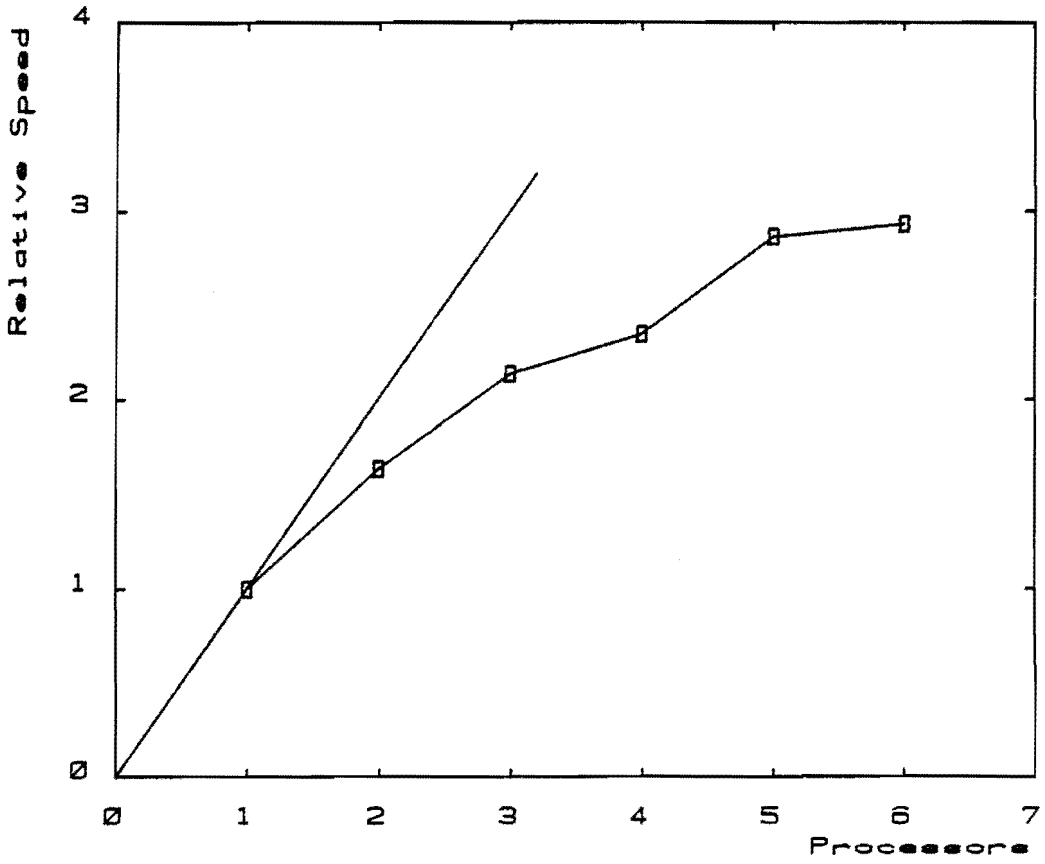


Figure 10.7: Performance for BGF Program Section - 24 Bus System

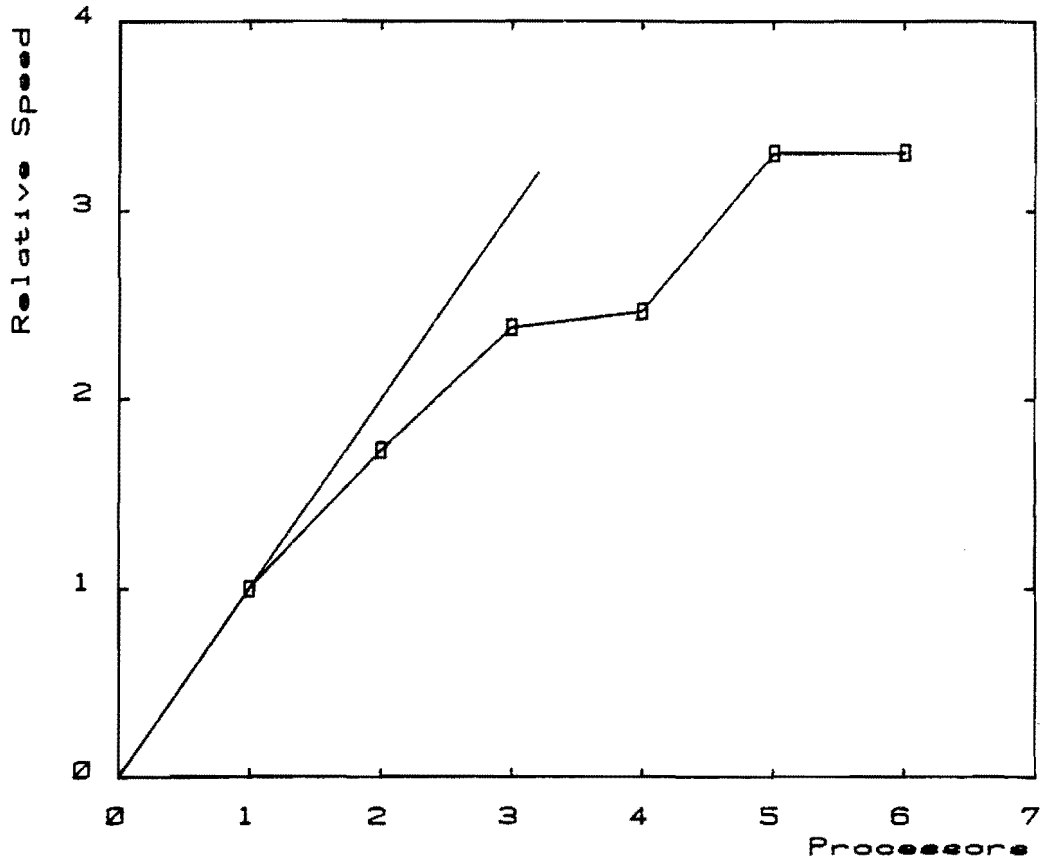


Figure 10.8: Performance for BG Program Section - 24 Bus System

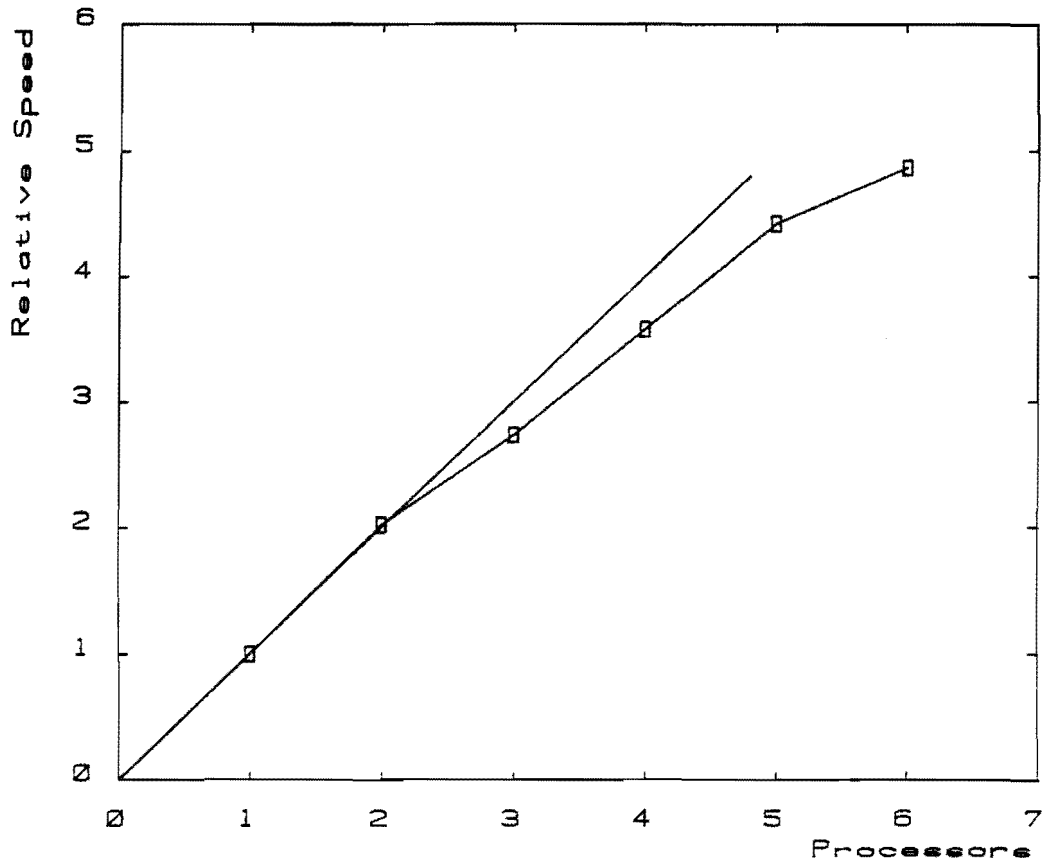


Figure 10.9: Performance for GF Program Section - 35 Bus System

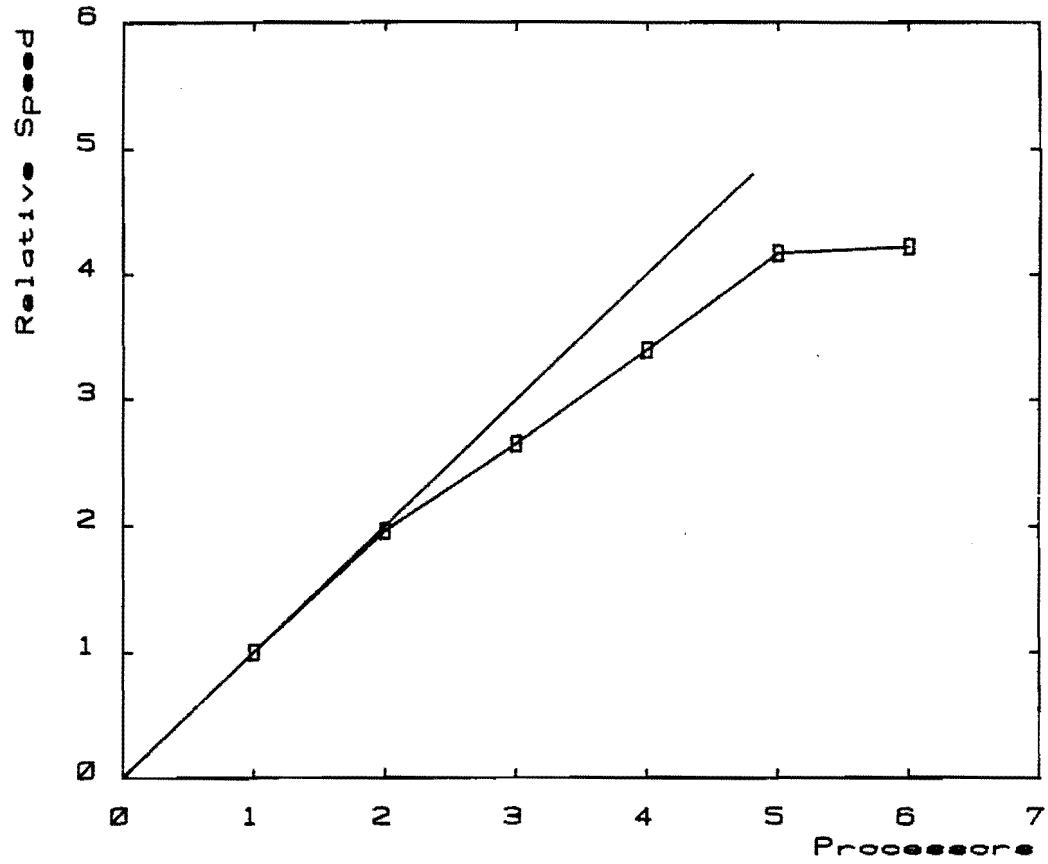


Figure 10.10: Performance for BGF Program Section - 35 Bus System

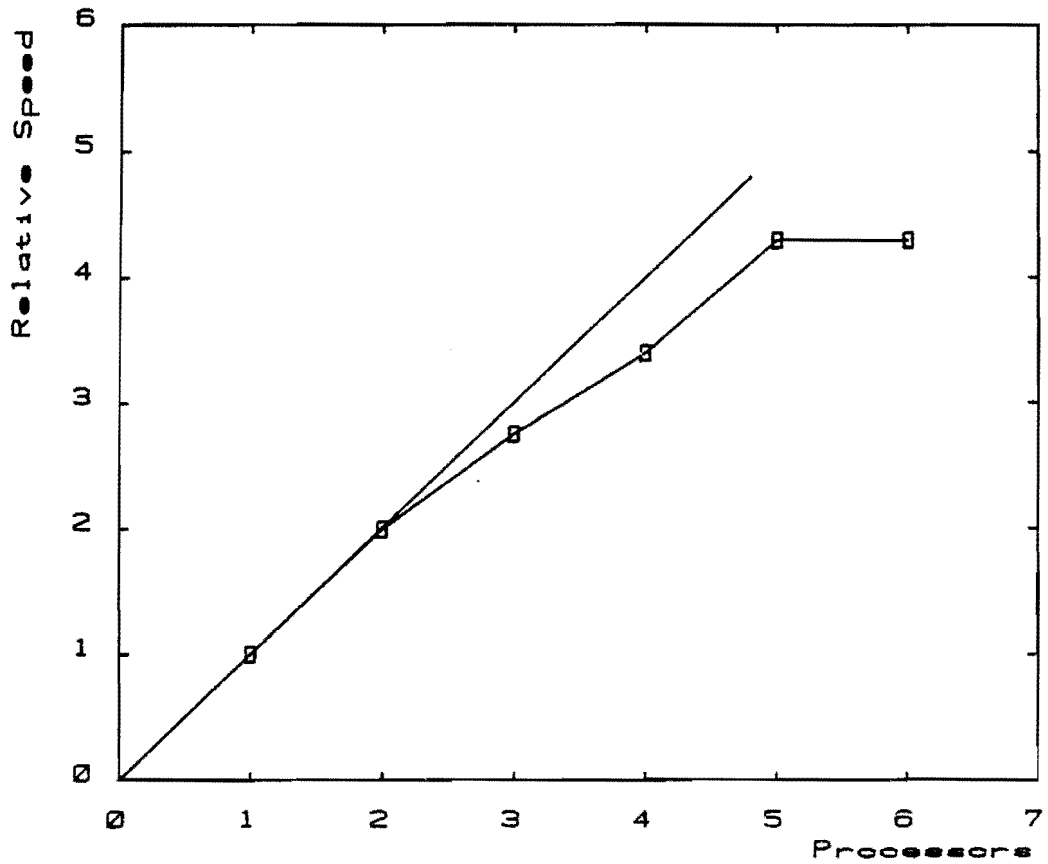


Figure 10.11: Performance for BG Program Section - 35 Bus System

processor operation is due to a problem which will be referred to as task availability aliasing:

. If four processors operate, and the five generators must be modelled, the distribution of tasks among processors may be similar to that depicted in figure 10.12. Because three processors are idle during the second half of the execution time, efficiency is low. The problem is most apparent where the number of tasks is slightly less than the number of processors or a small integral multiple thereof, and tasks require similar execution times.

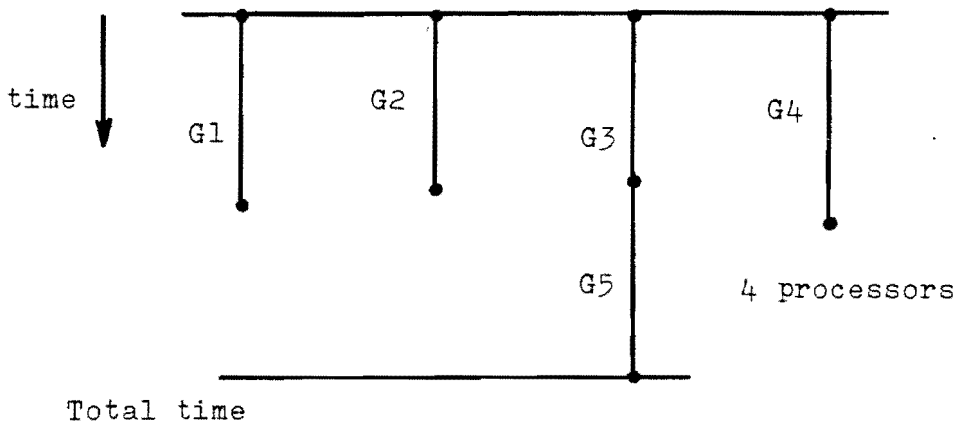


Figure 10.12: Five Tasks - Four Processors, an Illustration of Task Availability Aliasing

Task availability aliasing has no observable effect on the larger system where the number of generators greatly exceeds the number of processors.

10.7.2 Enhanced Scheduling Schemes

Performance characteristics for all code sections for both the network and non-network priority schemes are given in figures 10.13 to 10.18. In all cases there is a clear improvement over results achieved using the serial approach.

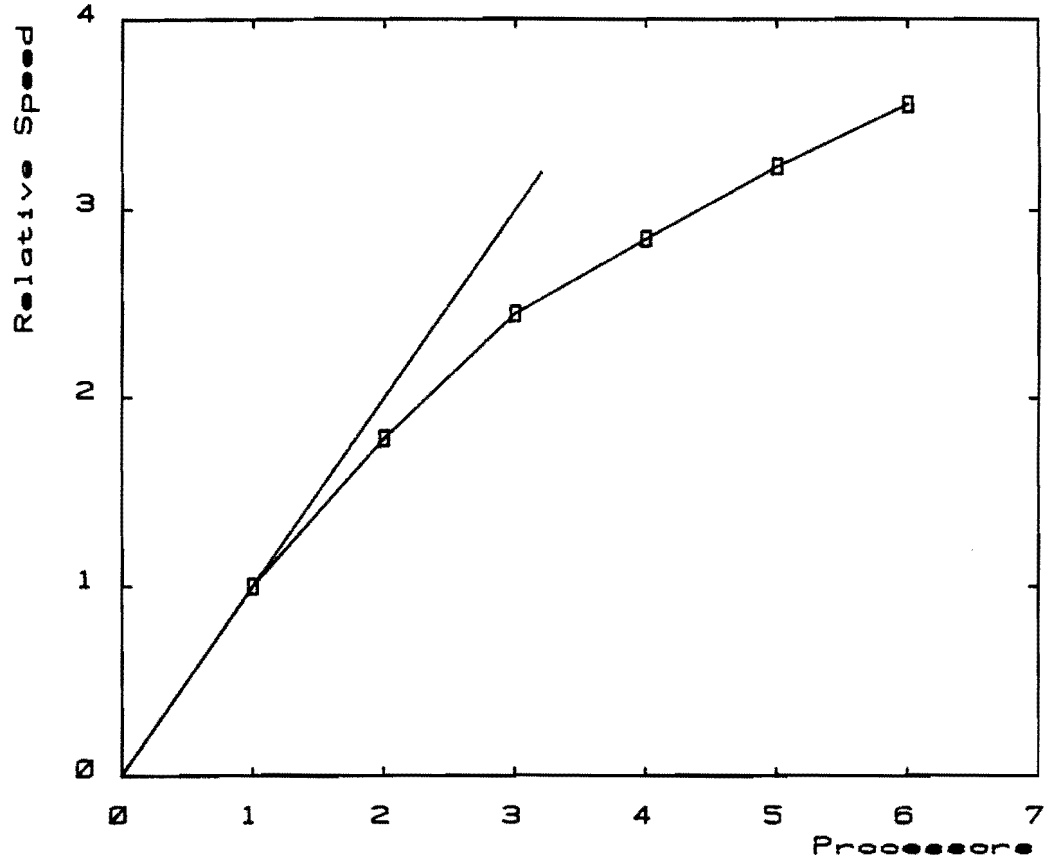


Figure 10.13: Performance with Non-Network Priority
(GF Section - 24 Bus System)

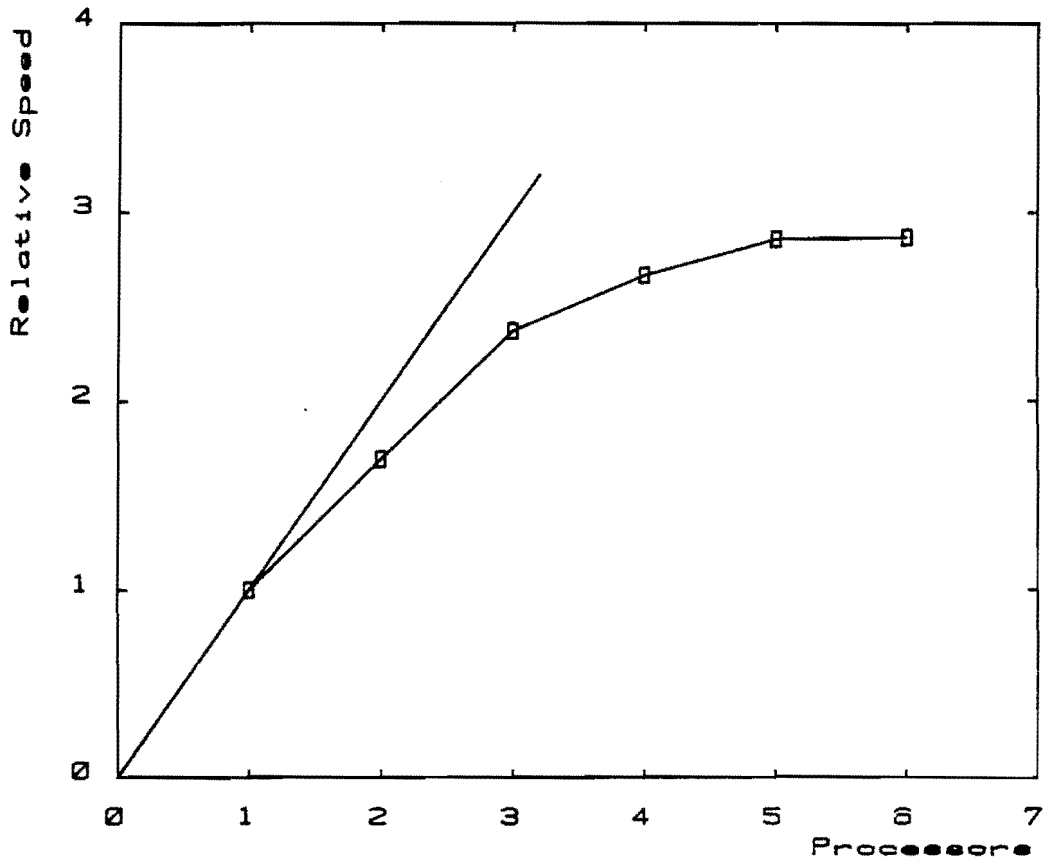


Figure 10.14: Performance with Non-Network Priority
(BGF Section - 24 Bus System)

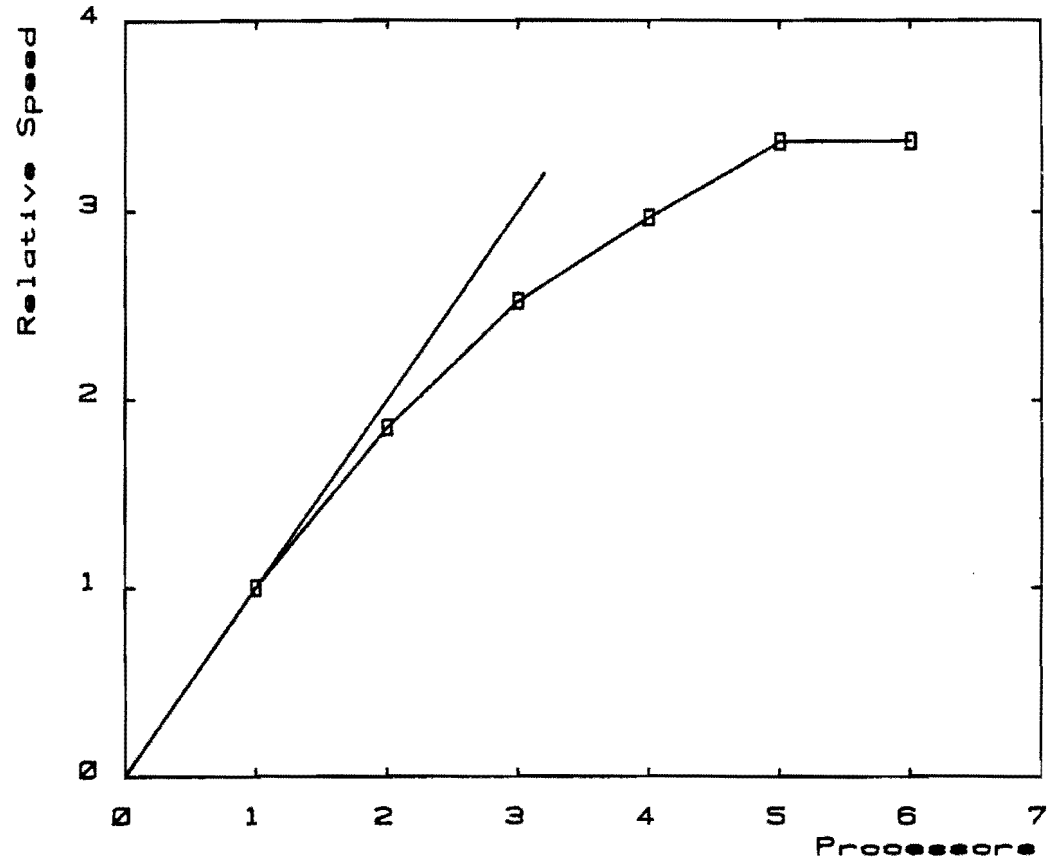


Figure 10.15: Performance with Non-Network Priority (BG Section - 24 Bus System)

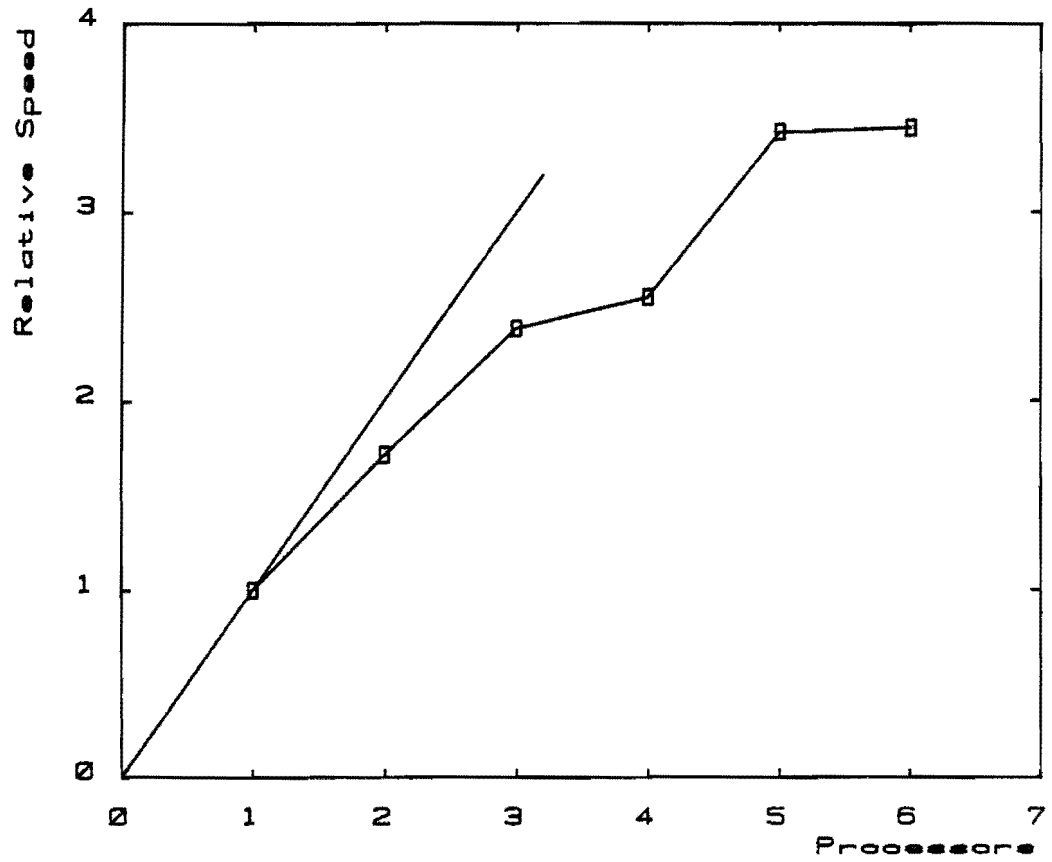


Figure 10.16: Performance with Network Priority (GF Section - 24 Bus System)

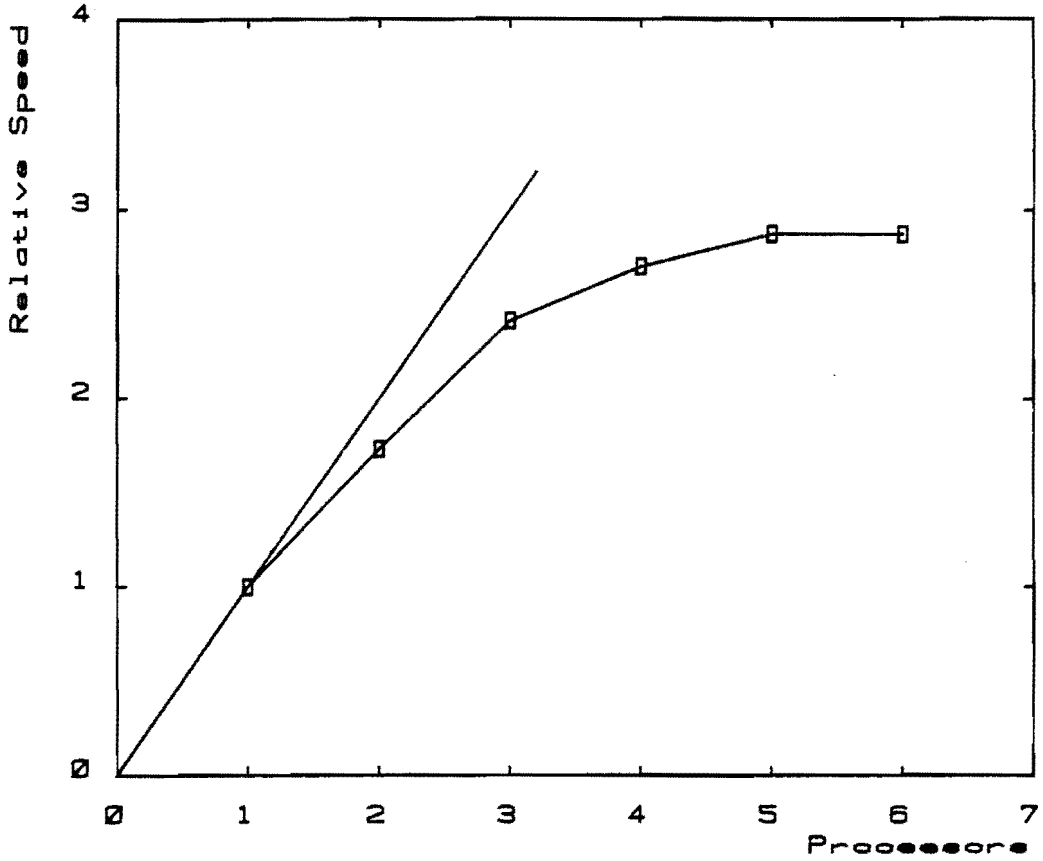


Figure 10.17: Performance with Network Priority (BGF Section - 24 Bus System)

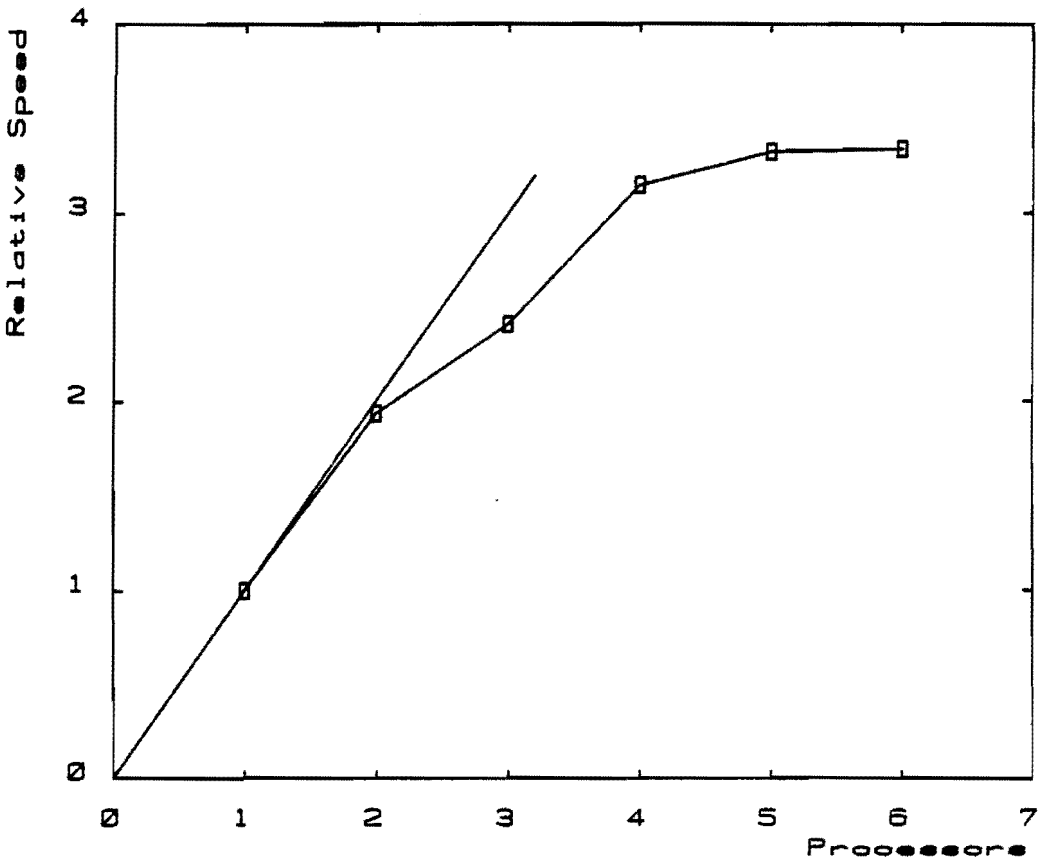


Figure 10.18: Performance with Network Priority (BG Section - 24 Bus System)

The most significant improvement is the reduction in the effect of task availability aliasing. The possibility of combining generator and network solutions may enable, for instance, the operation of the three idle processors in figure 10.12. Aliasing is not completely overcome, however, and is, for instance, still noticeable in figure 10.16.

In figure 10.19 the performance expected in executing a complete time step is given. Three iterations are assumed. It is clear that both the enhanced scheduling schemes are significantly more efficient than the serial approach. There is little evidence, however, of any significant difference between the enhanced approaches. When these three characteristics are compared with the dependent ideal curve (figure 10.3) the serial scheme is predictably below ideal while both the enhanced schemes are slightly, but consistently, above it. Hence, the gains made in utilising the independence of network and non-network related tasks more than offset the losses due to both the lack of total freedom to distribute generator models among processors and management processing.

10.8 Observations and Conclusions

The independence of non-network related tasks results in significantly improved performance in execution of the full transient stability analysis program when compared with the solution of linear equations only. In addition, utilisation of parallelism existing between network and non-network related models results in further performance improvement.

Three non-network task scheduling algorithms have been prepared and implemented. Of these, two overlap network and non-network related task executions. Both are based on predetermined task priorities in scheduling. The performance obtained by both approaches slightly exceeds the

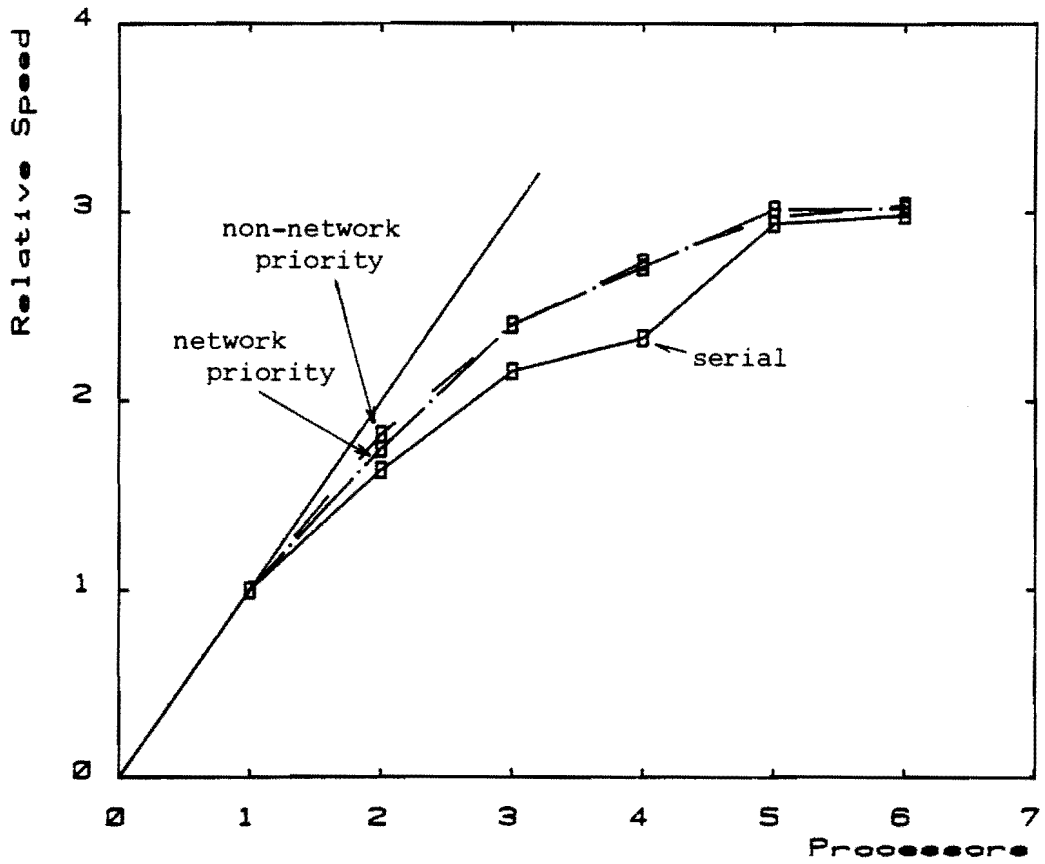


Figure 10.19: Comparison of Overall Performance When Using Serial and Enhanced Scheduling Schemes

constrained ideal performance characteristic.

No simulation of non-network related tasks has been developed. Therefore, no quantitative measure of management overheads is made. However, examination of the code involved suggests that these overheads are very low, especially because the execution times associated with non-network related tasks are comparatively long.

Absolute limits to the use of existing parallelism, such as that provided for linear solutions by Hu's method, are not known. The best estimate determined is the constrained ideal characteristic. However, it is still difficult to judge to what extent it is worthwhile pursuing

further performance improvement. Techniques which may be applied in attempting to attain better performance include:

.dynamic task priority management in scheduling algorithms, and

.network reordering to ensure a suitable distribution of non-network related tasks throughout each iteration.

These approaches would have to be developed with consideration of the overheads they introduce: in the first case in the form of increased management, and in the second as a result of fill-in of factor matrices due to changes from optimal sparsity related ordering.

In summary, straight forward approaches to non-linear task scheduling have been shown to result in good performance. Management overheads in these schemes are low, so more complex scheduling etc. could be of value. However, the degree to which performance is likely to improve, while not known, is expected to be small.

CHAPTER 11

CONCLUSIONS AND IDENTIFICATION OF FURTHER AREAS FOR RESEARCH

11.1 Conclusions

Various approaches to problems associated with the parallel implementation of transient stability analysis programs have been aimed at high utilisation of processing capability while using many processors. Multiple instruction multiple data stream (MIMD) processors are most suited to achieving this objective. However, different specification of requirements, such as economy, could mean single instruction multiple data stream (SIMD) systems were more appropriate.

In serial executions the great majority of processing time is spent in the execution of a loop which iterates between network and non-network models. Serial execution times in these two areas are similar but, because of much greater data dependence, the network solution represents a bottleneck in parallel implementation. Hence, opportunities for the most significant gains in performance arise in determinition of efficient methods for the parallel solution of linear equations.

Many schemes directed at the parallel solution of large sparse sets of linear equations have been described over the past few years. Theoretical limits to execution speed have not been approached by practical approaches until very recently. A new method, called the PBIF algorithm, has been developed. It is aimed at very efficient practical performance.

To test and validate the performance of the PBIF and other algorithms, a cost effective, research oriented multiprocessor, called the

UCMP system, was designed and built. Features aimed at efficient code execution include:

.sufficient processors to realistically assess parallel performance,

.bus structure and memory matched to the communication and storage needs of transient stability analysis,

.16-bit processors suited to transient stability analysis by possessing sufficient addressing capability and multiprocessing capability,

.ability to be simply extended to very high raw processing speed through the use of floating point coprocessors, and

.built in accurate performance measuring timers.

To enable both efficient code development and quick performance evaluation, the UCMP system can operate in either of two modes:

.stand alone, possibly with individual processors replaced by an in-circuit emulator. This mode is aimed at code development.

.under control of, and in communication with, a host computer. This mode is intended for fast testing using, for instance, a number of power system descriptive data sets.

Features improving operation during both program development and performance measurement include:

.the use of commercially supplied high level languages and symbolic serial debuggers,

.special hardware enabling effective debugging in a multiprocessing environment,

.non-crucial serial execution can be performed by the host computer,

.access to the mass storage and the complex, flexible operating system of the host computer, and

.high speed communication with the host computer.

Operation is supported by a flexible set of software utilities which include facilities for simple changes between modes. The user is presented with a variety of options from which he may choose the most appropriate at any stage in development. Although some knowledge of the system in needed for operation, many details are transparently handled by the software utilities.

The UCMP system and a detailed simulation were used to evaluate the execution performance of the PBIF algorithm. Using microprocessors, performance approaching the theoritical limits was found to be possible over the full range of processor numbers. The practical performance achieved is, in fact, better than any other known methods. However, judgement of performance is based on assumptions with respect to the capabilities of inter-processor communication networks, a feature which varies greatly between multiprocessing systems. In some cases, it is possible that schemes other than that used in the PBIF algorithm will execute more efficiently due to lower communication requirements.

A problem related to the layout of networks was found to degrade performance. The extent of degradation varied from system to system and, although significant, was not overwhelming for the larger power systems

used as benchmark examples. Both the level of execution speed reduction and methods of overcoming the problem were considered.

To evaluate the possible execution performance of the complete transient stability analysis program all significant program sections were implemented on the UCMP system. Results achieved showed insignificant degradation due to lack of parallelism in the non-network component model solutions. This confirmed the expectation that the linear equation solutions provide a bottleneck. Tests were made using schemes exploiting the parallelism existing between the network and non-network related tasks. Two task scheduling methods were tried. Performance of both was almost identical, and it was significantly better than that where this parallelism was not utilised.

Combining the execution performance of linear equation solutions, both real and simulated, with that of the complete transient stability analysis program, it can be concluded that speed up by a factor of 100 is feasible. If this was reached using currently available high performance processors then stability of large power systems could be analysed in real time. The greatest difficulty forseen in achieving this is the selection and creation of an appropriate processor-memory interconnection network. An acceptable busing arrangement would probably be possible but very expensive. Note that hardware suited to the modular construction of multiple bus systems is becoming commercially available.

If very high execution speed is not required, but fast, cost effective analysis is desirable, then the UCMP system approach is directly applicable. Power authorities with existing suitable host computers could obtain MFLOP performance by adding, for example, hardware floating point coprocessors. The current retail cost of the hardware components of the UCMP system would be approximately $NZ21,000.

Many avenues for further investigation remain in this new field. It is clear when viewing the expanding, but already widespread, interest in the parallel execution of transient stability analysis programs, especially by research coordinating bodies such as the Electric Power Research Institute (EPRI), that developments in the area, if successful, will play a significant role in the future of power systems analysis.

11.2 Suggestions for Further Research

With respect to the work presented in this thesis, two major areas are identified in which effort, aimed at improving the performance of multiprocessors in transient stability analysis, would be most cost effective are:

.appropriate hardware choice, and

.further tuning of programs.

More detailed consideration of hardware interconnection arrangements would not be of value until a particular requirement was specified. For development of a very fast practical system, the busing scheme would be an early and very important consideration.

Increases in program execution efficiency could result from further investigation in the following areas:

.Elemental approaches to linear equation solutions appear most efficient, but detailed comparison of task scheduling schemes, taking account of the hardware environment, could be valuable. It is possible that different schemes would provide better performance over different ranges of processor number.

.Methods to execute program sections outside the central

loop in parallel may become important when using very high numbers of processors. Note that in an on-line environment (a) input and output sections will differ greatly when compared with off-line execution, and (b) the same system is analysed every execution so program sections, such as matrix ordering, could be pre-processed.

.Optimal network nodal ordering, reducing performance degradation due to nodal distribution ill-conditioning is hampered by dependence on the number of processors. Therefore, it is probably best considered initially with a fixed number of processors.

.The amount of scope existing for exploitation of parallelism between the network and non-network related solutions has not been established. It is possible that further gains in execution speed could be achieved through utilisation of this parallelism but, because the linear equation solution bottleneck still exists, improvements are likely to be small.

REFERENCES

ACKERMAN, W.B., 1982.
"Data Flow Languages",
IEEE Computer, Vol. 15, No. 2, pp. 15-25, Feb. 1982.

ADAMS, G.B., III and SIEGEL, H.J., 1982.
"On the Number of Permutations Performable by the Augmented Data Manipulator Network",
Trans. IEEE, Vol. C-31, pp. 270-77, Apr. 1982.

AGERWALA, T., and ARVIND, 1982.
"Data Flow Systems: Guest Editors' Introduction",
IEEE Computer, Vol. 15, No. 2, pp. 10-3, Feb. 1982.

ALVARADO, F.L., 1979.
"Parallel Solution of Transient Stability Problems by Trapezoidal Integration",
Trans. IEEE, Vol. PAS-93, pp. 1080-90 May/June 1979.

AMC - 1980.
"AM95/4010 Monoboard Computer",
Product Description, Advanced Micro Computers, copyright 1980.

AMD - 1980.
"MOS/LSI Data Book",
Advanced Micro Devices Inc., 1980.

ARNOLD, C.P., 1976.
"Solution of the Multimachine Power System Stability Problem",
University of Victoria, Manchester, England,
(Thesis: Ph.D.: Engineering).

ARNOLD, C.P., PARR, M.I., AND DEWE, M.B., 1983.
"An Efficient Parallel Algorithm for the Solution of Large Sparse Linear Matrix Equations",
to be published in IEEE Trans. Computers, (see App. 6)

BAILEY, D.G., 1981.
"Software Interface for Interprocessor Communication",
Final Year Project Report, Dept. of Electrical Engineering,
University of Canterbury, 1981.

BARRY, D.E., 1978.
"Technology Assessment Study of Near Term Computer Capabilities and Their Impact on Power Flow and Stability Simulation Programs",
EPRI EL-946, TPS 77-749, Final Report, Dec., 1978.

BARTH, M.J., 1981 (a).
"Development of Microprocessor Based Protection Relay Modules",
University of Canterbury, Christchurch, New Zealand, 1981,
(Report: M.E.: Engineering)

BARTH, M.J., 1981 (b).
"SBC UC/86 Single Board Computer Reference Manual",
Dept. of Electrical Engineering, University of Canterbury, 1981.

BARTH, M.J. 1981 (c).
"128K Byte Dynamic RAM 16K Byte EPROM MULTIBUS Board Reference Manual",
Dept. of Electrical Engineering, University of Canterbury, 1981.

BRAMELLER, A., ALLAN, R.N. and HAMAM, Y.M., 1976.
"Sparsity,"
Pitman Publishing, 1976.

BRASCH, F.M.(Jr.), VAN NESS, J.E. and SANG-CHUL KANG, 1978.
"Evaluation of Multiprocessor Algorithms for Transient Stability Problems",
EPRI EL-947, Technical Planning Study 77-718, Nov. 1978.

BRASCH, F.M.(Jr.), VAN NESS, J.E. and SANG-CHUL KANG, 1979.
"The Use of a Multiprocessor Network for the Transient Stability Problem",
Proc. 1979 Power Industry Computer Applications Conference, Cleveland, OH, pp. 337-44, May 1979.

BRASCH, F.M.(Jr.), VAN NESS, J.E. and SANG-CHUL KANG, 1981.
"Design of Multiprocessor Structures for Simulation of Power System Dynamics",
EPRI EL-1756, Research Project 1355-1, Final Report, Mar. 1981.

BRASCH, F.M.(Jr.), VAN NESS, J.E. and SANG-CHUL KANG, 1982.
"Simulation of a Multiprocessor Network for Power System Problems",
Trans. IEEE, Vol. PAS-101, No. 2, pp. 295-01, Feb. 1982.

BRINKMAN, B., DOWSON, M., McBRIDE, B. and SMITH, G., 1980.
"The DEMOS 86 Multiple Processor Computer",
Scicon Consultancy International Ltd., 1980.

BROWN, E.P.M., 1981.
"Power System State Estimation and Probabalistic Load Flow Analysis",
University of Canterbury, Christchurch, New Zealand, 1981,
(Thesis: Ph.D.: Engineering)

BURROUGHS - 1975.
"B6700 Timings for Selected Languages Constructs (relative to Mark 2.6 software release)",
Burroughs form No. 5000854, Feb. 1975.

BYERLY, R.T. and KIMBARK, E.W. (editors), 1974.
"Stability of Large Electric Power Systems",
IEEE Press, 1974.

COMPUTER ADVTS., 1981.
advertisements appearing in IEEE Computer,
Vol. 14, No. 9, Sept. 1981.

CONCORDIA, C. and SCHULZ, R.P., 1975.
"Appropriate Component Representation for the Simulation of Power System Dynamics",
IEEE PES, 1975 Winter Meeting, Symposium on Adequacy and Philosophy of Modeling: Dynamic System Performance, New York, pp. 16-23.

CONRAD, V. and WALLACH, Y., 1977.
"Iterative Solution of Linear Equations on a Parallel Processor System",
Trans. IEEE, Vol. C-26, No. 9, pp. 838-47, Sept. 1977.

DEC(a) - 1979.
"Microcomputer Processor Handbook",
Digital Equipment Corp., 1979-80.

DEC(b) - 1978.
"PDP-11 Peripherals Handbook",
Digital Equipment Corp., 1978.

DEC(c) - 1975.
"DR11-K Interface User's Guide and Maintenance Manual",
Digital Equipment Corp., 1975

DEC(d) - 1978.
"VAX-11 Software Handbook",
Digital Equipment Corp., 1978.

DEC(e) - 1980.
"VAX-11 FORTRAN User's Guide",
Digital Equipment Corp., Order No. AA-D035B-TE, Apr. 1980.

DEC(f) - 1981.
"Microcomputers and Memories",
Digital Equipment Corp., 1981.

DEMINET, J., 1982.
"Experience with Multiprocessor Algorithms",
Trans. IEEE, Vol. C-31, pp. 278-88, Apr. 1982.

DEO, N., 1974.
"Graph Theory with Applications to Engineering and Computer Science",
Prentice-Hall, 1974.

DOMMEL, H.W. and SATO, N., 1972.
"Fast Transient Stability Solutions",
Trans. IEEE, Vol. PAS-91, No. 4, pp. 1643-50, July-Aug, 1972.

DUGAN, R.C., DURHAM, I., and TALUKDAR, S.N., 1979.
"An Algorithm for Power System Simulation by Parallel Processing",
Conference Paper A79 442-5, IEEE Summer Power Meeting,
Vancouver, Canada, July 15-20 1979.

DURHAM, I., DUGAN, R.C., JONES, A.K., and TALUKDAR, S.N., 1979.
"Power System Simulation on a Multiprocessor",
Conference Paper A79 487-2, IEEE Summer Power Meeting,
Vancouver, Canada, July 15-20 1979.

EL GUINDI, M. and MANSOUR M., 1982.
"Transient Stability of a Power System by the Liapunov Method Considering the Transfer Conductances",
Trans. IEEE, Vol. PAS-101, No. 5, pp. 1088-93, May 1982.

ENSLOW, P.H., Editor, 1974.
"Multiprocessors and Parallel Processing",
Comptre Corporation, Wiley-Interscience, John Wiley and sons, New York, 1974.

FAIRBOURN, D.G., 1982.
"VLSI: A New Frontier for System Designers",
IEEE Computer, Vol. 15, No. 1, pp. 87-96, Jan. 1982.

FAWCETT, J. and BICKART, T.A., 1980.
"Cellular Arrays in the Solution of Large Sets of Linear Equations",
International Conference on Circuits and Systems, Port Chester, NY., pp. 984-87, 1980.

FONG J. and POTTLE, C., 1978.
"Parallel Processing of Power System Analysis Problems via Simple Parallel Microcomputer Structures",
IEEE Trans PAS., Vol. PAS-97, pp. 1834-41, Sept/Oct. 1978.

FONG, J., 1978.
"Large Scale Power System and Nonlinear Network Simulation Via Simple Parallel Microcomputer Structures",
Cornell University, 1978, (Thesis: Ph.D.: Engineering).

HAPP, H.H., POTTLE, C. and WIRGAU, K.A., 1978.
"Parallel Processing for Large Scale Transient Stability",
IEEE Canadian Conf. Comm. and Power,
Conference Paper 78 CH 1373-0 REG 7, pp. 204-7, 1978.

HAPP, H.H., POTTLE, C. and WIRGAU, K.A., 1979.
"An Evaluation of Present and Future Computer Technology for Large Scale Power System Simulation",
IFAC Int. Symp. on Computer Applications in Large Power Systems, New Delhi, India, Aug. 1979.

HAPP, H.H., POTTLE, C. and WIRGAU, K.A., 1979.
"An Assessment of Computer Technology for Large Scale Power System Simulation",
Proc. 1979 Power Industry Computer Applications Conference, Cleveland, OH, pp. 316-24, May 1979.

HATCHER, W.L., BRASCH, F.M.(Jr.) and VAN NESS, J.E., 1977.
"A Feasibility Study for the Solution of Transient Stability Problems by Multiprocessor Structures",
Trans. IEEE, Vol. PAS-96, No. 6, pp. 1789-97, Nov/Dec 1977.

HAYNES, L.S., LAU, R.L., SIEWIOREK, D.P. and MIZELL, D.W., 1982.
"A Survey of Highly Parallel Computing",
IEEE Computer, Vol. 15, No. 1, pp.9-24, Jan. 1982.

HUANG, J.W. and WING, O., 1978.
"On Minimal Completion Time and Optimal Scheduling of Parallel Triangulation of a Sparse Matrix",
Conference Paper Ref. CH1361-5/78/0000-5670, IEEE Summer Power Meeting, L.A., CA., July 1978.

INTEL(a) - 1982.
"Component Data Catalog",
Intel Corp., Order No. 210298-001, Jan. 1982.

INTEL(b) - 1982.
"Systems Data Catalog",
Intel Corp., Order No. 210299-001, Jan. 1982.

INTEL(c) - 1978.
"iSBC 86/12 Single Board Computer Hardware Reference Manual",
Intel Corp., Order No. 9800645A, 1978.

INTEL(d) - 1978.
"Multibus Specification",
Intel Corp., Order No. 9800683, 1978.

INTEL(e) - 1978.
"MCS-85 User's Manual",
Intel Corp., Order No. 9800366D, 1978.

INTEL(f) - 1979.
"MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users",
Intel Corp., Order No. 9800639-03, 1979.

INTEL(g) - 1979.
"The 8086 Family User's Manual",
Intel Corp., Order No. 9800722-03, Oct. 1979.

INTEL(h) - 1980.
8086 Microprocessor Seminar Notes,
Intel Corp., 1980.

INTEL(i) - 1978.
"PL/M-86 Programming Manual",
Intel Corp., Order No. 9800466A, 1978.

INTEL(j) - 1979.
"MCS-86 Macro Assembly Language Reference Manual",
Intel Corp., Order No. 9800640-02, 1979.

INTEL(k) - 1979.
"ICE-86 In-Circuit Emulator Operating Instructions for ISIS-II Users",
Intel Corp., Order No. 9800714A, 1979.

INTEL(l) - 1979.
"8080/8085 Floating-Point Arithmetic Library User's Manual",
Intel Corp., Order No. 9800452-03, 1979.

KEES, H.G.M. and JESS, J.A.G., 1980.
"A Study on the Parallel Organisation of the Solution of a 1600-Node Network",
Conference Paper Ref. CH 1511-5/80/0000-0988, International Conference on Circuits and Systems, Port Chester, NY., pp. 988-91, 1980.

KUNG, H.T., 1982.
"Why Systolic Architectures?",
IEEE Computer, Vol. 15, No. 1, pp. 37-46, Jan. 1982.

LOW, W.C., 1980.
"A DMA Interface Controller",
Final Year Project Report, Dept. of Elecrical Engineering, University of Canterbury, 1980.

LUNDSTROM, S.F. and BARNES G.F., 1980.
"A Controllable MIMD Architecture",
Proc. 1980 International Conf. on Parallel Processing, Harbor Springs, Michigan, pp. 19-27.

MOT - 1981.
"Motorola 68000 Course Notes",
Motorola Technical Training, Phoenix, Arizona, Mar. 1981.

OREM, F.M. and TINNEY, W.F., 1979.
"Evaluation of an Array Processor for Power System Applications",
Proc. 1979 Power Industry Computer Applications Conference, Cleveland, OH, pp. 345-50, May 1979.

PARR, M.I., 1980.
"MULTIBUS Backplane Description",
(Documentation of the backplane developed for use in the UCMP system), Dept. of Electrical Engineering, University of Canterbury, Nov. 1980.

PATEL, J.H., 1981.
"Performance of Processor-Memory Interconnections for Multiprocessors",
Trans. IEEE, Vol. C-30, No. 10, pp. 771-80, Oct. 1981.

PODMORE, R., LIVERIGHT, M., VIRMANI, S., PETERSON, N.M., and BRITTON,J., 1979.
"Application of an Array Processor for Power System Network Computations",
Proc. 1979 Power Industry Computer Applications Conference, Cleveland, OH, pp. 325-31, May 1979.

POTTLE, C., 1980.
"The Use of an Attached Scientific ("array") Processor to Speed Up Large-scale Power Flow Simulations,"
Conference Paper Ref. CH 1511-5/80/0000-0980, International Conference on Circuits and Systems, Port Chester, NY., pp. 980-3, 1980.

RUSSELL, R.M., 1978.
"The CRAY-1 Computer System",
Communications of the ACM, No. 1, pp.63-72, Jan. 1978.

SATYANARAYANAN, M., 1980.
"Multiprocessors: A Comparative Study",
Prentice-Hall Inc., 1980.

SAUNDERS, G.D., 1979.
"Power Systems Transient Stability Simulation",
Final Year Project Report, Dept. of Elecrical Engineering,
University of Canterbury, 1979.

SHIMOR, A. and WALLACH, Y., 1978.
"A Multibus-Oriented Parallel Processor System",
Trans. IEEE, Vol. IECI-25, pp.137-40, May 1978.

SNYDER, L., 1982.
"Introduction to the Configurable, Highly Parallel Computer",
IEEE Computer, Vol. 15, No. 1, pp.47-56, Jan. 1982.

WALLACH, Y. and CONRAD, V., 1980.
"On Block-parallel Methods for Solving Linear Equations",
Trans. IEEE, Vol. C-29, pp. 354-59, May 1980.

WATSON, I. and GURD, J., 1982.
"A Practical Data Flow Computer",
IEEE Computer, Vol. 15, No. 2, pp.51-7, Feb. 1982.

WING, O. and HUANG, J.W., 1980.
"A Computational Model of Parallel Solution of Linear Equations",
Trans. IEEE, Vol. C-29, pp. 632-8, July 1980.

ZOLLENKOPF, K., 1971.
"Bifactorisation: Basic Computational Algorithm and Programming
Techniques", in "Large Sparse Sets of Linear Equations",
pp.75-96, Academic Press,1971.

APPENDIX 1

GLOSSARY

Computer Oriented Terms

assembler - a translator for low level, assembly, languages

backplane - a set of conductors and circuit elements which is intended to connect, in an organised way, the signal lines of a number of printed circuit boards

board - a printed wiring assembly on which ICs etc. are mounted

bus - a group of conductors used for transmitting signals or power from one or more sources to one or more destinations

bus master - a computational element capable of asserting commands on a bus

bus saturation - condition existing when a commonly used bus is continuously busy, and thus impedes the operation of processors

byte - a group of eight bits

code - a set of symbols providing information in a form suited to execution by a processor

compiler - a translator for high level languages

concurrent - 'pertaining to the occurrence of two or more events or activities within the same specified interval of time' (Enslow, 1974, p.133)

core components - computer components necessary for execution of application, rather than support, programs

current bus master - a computational element presently accessing a bus

data - operands and results involved in any operation or set of operations

dependent - requiring the result/s of one or more other operations

dynamic - varying during program execution

dynamic task states - (defined in detail in section 2.4.1)

- not ready
- ready
- running
- completed

entry point - address at which execution of a program should begin

execution address - the address at which code and data are seen by the processor executing a program

hibernation - a state adopted by a processor during which it relinquishes use of any commonly accessed bus and cannot affect the state of any memory

highly parallel - refers to multiprocessors with many individual processors

homogeneous - a multiprocessor in which all processors have a similar view of globally accessible memory, and experience similar delays in accessing it

load address - the address at which information is viewed by the element loading code, (for subsequent execution), and data

machine code - code used to represent the elementary operations of a programming system

operating system - software which coordinates the operation of a computer, and aids the user in utilising available resources

parallelism - the degree to which a program can be distributed effectively among a number of processors

polymorphism - ability to be reconfigured to perform different tasks

run time - the period during which a program is executed

semaphore - a bit in memory with a state which indicates the availability of a commonly updatable resource

serial optimal - a programming method which is very efficient in serial execution

simultaneous - 'pertaining to the occurrence of two or more events existing or occurring at the same instant of time' (Enslow, 1974, p.137)

source code - code prepared by a programmer, intended as input to a translator

stack - a group of memory locations, utilised by a number of specially oriented instructions, which make access to that memory on a last-in, first-out basis

support components - computer elements, other than core components, required for development and testing of application programs

systolic system - a system in which information 'flows between cells in a pipelined fashion, and communicates with the outside world only occurs at the "boundary cells"' (Kung, 1982)

translator - a program which produces machine executable code after translation from source code

VAX - a Digital Equipment Corporation VAX 11/780 minicomputer

VAX/VMS - the VAX computer operating system which is used in the Dept. of Electrical Engineering

word - a group of bits. Throughout this thesis 16 bit wordlength is assumed

Power System Oriented Terms

bus - a conductor, or group of conductors, that serve as a common connection of two or more circuits

cut set - elements of a matrix which describe the connections between groups of nodes

fill-in - the introduction of new elements during matrix factorisation

factorisation - the formation of factor matrices required in, for instance, the bifactorisation method

infinite system - a point within a power system at which the voltage is not affected by any changes of, for instance, load conditions

Jacobian - a matrix of differential elements required in Newton-Raphson methods

sparsity coefficient - 'the ratio between the number of zero elements and the total number of elements in a matrix' (Brameller et al, 1976, p.21)

synchronous position - the location of a synchronous machine's rotor within a synchronously rotating frame of reference

APPENDIX 2

MULTIPROCESSOR PERFORMANCE MEASURES

The performance of a parallel processing system can be expressed in many ways. The use of inappropriate measures can be misleading in presentation of the effectiveness of a system. A number of possible measures are defined in this appendix with the objective of clarification of the range of possibilities.

Execution speed can be measured as an absolute rate, for instance on a per second basis, or can be related to the performance of a single processor.

All of the measures presented vary as a function of the number of processors. The form of this variation can itself be a measure of multiprocessor performance.

Although terminology such as 'speed up' and 'MFLOPS' are commonly used, the nomenclature has not been standardised.

Absolute performance measures include:

Execution time: the period required to run a program

MFLOPS: the rate at which instructions are executed - in this case millions of floating point operations per second

effective MFLOPS: the rate at which useful instructions are executed. Useful refers to instructions not involved, for instance, in the management processing implied by parallel execution

Performance measures related to single processor operation are:

Speed up*: the factor by which execution time is reduced in comparison with single processor execution of a program

Effective number of processors, relative speed: same as speed up

Processing efficiency: speed up divided by the number of processors involved in program execution

---

* - Note that the most common method used to depict performance throughout this thesis is to express speed up as a function of the number of processors.

APPENDIX 3

SUBSTITUTION STEPS IN THE LU AND BIFACTORISATION METHODS OF SOLUTION OF LINEAR EQUATIONS

Both the LU factorisation and bifactorisation techniques are systematic implementations of Gauss eliminition. They are particularly suited to organised execution on digital computers, especially when sparse matrices are involved. A full description of all these methods, including the factorisation steps, is given by Brameller et al (1976).

Although the numerical values differ, the number and order of computations throughout the execution of the substitution steps in both techniques are identical. The steps involved in implementation of bifactorisation are less amenable to simple, compact explanation than those in LU factorisation. Consequently, although bifactorisation has been used in the programs implemented, LU factorisation is selected as the most suitable basis for descriptions presented in the text of this thesis.

LU Factorisation

In the solution of x in the equation

$$Ax=b$$

A can be expressed as the product of two factor matrices:

$$A=LU$$

where:

L is a lower triangular matrix ie. it has no non-zero elements above the diagonal.

U is an upper triangular matrix with unity elements on its diagonal.

Once the L and U factors are found by triangulation, the solution steps in finding x are as follows. (Note that throughout the substitution process, a vector which has an initial value of b and a final value of x is used. This vector is called z.)

1. to find an intermediate vector z by forward substitution

   $Lz=b$; and similarly

2. to find the solution x by backward substitution

   $Ux=z$

Bifactorisation

For an n-th order problem, the inverse of A can be expressed as the product of 2n factor matrices.

$$R_1 \; R_2 \; . \; . \; . \; R_n \; S_n \; S_{n-1} \; . \; . \; S_1 = A^{-1}$$

The solution, x, can therefore be found by a series of calculations of the product of a factor matrix and a vector.

$$x = \bar{A}^1 b$$

$$= R_1 \; . \; . \; . \; R_n \; S_n \; . \; . \; . \; S_1 \; b$$

The form of the factor matrices is:

S factors - unity diagonal, with zero elements elsewhere, except for the nth column which can contain both a diagonal and lower triangular elements.

R factors - unity diagonal, with zero elements elsewhere, except for the nth row which can contain off-diagonal upper triangular elements

Note that topologically the positions of the elements in all the S factors correspond to positions of elements in the L matrix. The same is true of the R factors and the U matrix.

APPENDIX 4

DEFINITION OF VECTORS USED IN ALGORITHM IMPLEMENTATION

ELE: The elements of the L and U matrices

POSN$1ST$FOR: position within ELE of the diagonal element in each column for forward substitutions

POSN$1ST$BACK: similar for back substitutions, but off-diagonal

NUMBERINCOLUMN$FOR: number of elements in each column for forward substitutions

NUMBERINCOLUMN$BACK: similar for back substitutions

NUMBERINROW$FOR: number of elements in each row for forward substitutions

NUMBERINROW$BACK$BACK: similar for back substitutions

B: elements of z

POSNINB$BEFORE$ORDERING: maps the present locations of elements within the B vector to the positions they were in before ordering

B$SEM: semaphores corresponding to both elements of B and columns within factor matrices. Separate sets of semaphores could be used.

BACK$L: vector to map to indices of ELE for back substitutions

ROW$NUMBER$BACK: row numbers of elements within ELE

APPENDIX 5

CODED IMPLEMENTATION OF THE BGF PROGRAM SECTION

The following routines implement the selection of tasks during the BGF program section ie. for the non-network and both the forward and backward substitution tasks. The procedure BACKWARDGENFORWARD coordinates searches for tasks and calls routines implementing forward substitutions (FORSUB), backward substitutions (BACKSUB), and non-network component models (NETTOGEN, GENTONET, and TRAP). Note that the routine SEARCH was presented in figure 4.7.

```
/* BGF task search coordination with network priority */
BACKWARD$GEN$FORWARD : PROCEDURE;
DECLARE (TRAP$STATE,BACK$STATE) BYTE;

        DO FOREVER;
                /* search for a back substitution task */
                IF (BACK$STATE:=SEARCH$BACK) = STILL$GOING THEN DO;
                        CALL BACKSUB;
                        BACK$SUBST$COUNT = BACKSUBST$COUNT - 1;
                        END;
                /* else search for a non-network related task */
                ELSE IF (TRAP$STATE:=SEARCH$TRAP) = STILL$GOING THEN DO;
                        CALL NET$TO$GEN;
                        CALL TRAP;
                        CALL GEN$TO$NET;
                        /* prepare for forward substitutions */
                        NUMBER$IN$ROW$FOR(KBUS(GENERATOR$NUMBER)) =
                            NUMBER$IN$ROW$FOR(KBUS(GENERATOR$NUMBER)) - 1;
                        DECREMENT$COMPLETED$COUNT;
                        END;

                /* if all backward substitution and non-network related
                        tasks have been allocated then go on to forward
                        substitutions */
                IF ((BACK$STATE=ENDED) AND (TRAP$STATE=ENDED)) THEN DO;
                        /* wait till backward substitutions completed */
                        DO WHILE BACKWARD$DONE <> TRUE; END;
                        GOTO BGF1;
                        END;
                END;

        BGF1:
        /* search for and implement forward substitutions. Note that
                other processors could still be involved in non-network
                related tasks */
        DO WHILE SEARCH <> ENDED;
                CALL FORSUB;
                DECREMENT$COMPLETED$COUNT;
                END;

        /* wait till all forward substitutions completed */
        DO WHILE FORWARD$DONE <> TRUE; END;

        END BACKWARD$GEN$FORWARD;
```

```
        /* To find a generator related task */
SEARCH$TRAP: PROCEDURE BYTE;
DECLARE (NUMBER,I) INTEGER;

        I = 0;

        DO WHILE I < 20;          /* try only the next 20 */
                IF ((NEXT$TRAP > KG) OR (GEN$DONE = TRUE))
                        THEN RETURN ENDED; /* exit if all done */

                IF ( I + NEXT$TRAP ) > KG THEN RETURN TASK$NOT$FOUND;
                /* exit if no ready task found among those left */
                J = I + NEXT$TRAP;

                PSEM TRAP$SEM(J) SET_;
                /* test if all predecessors completed and this task not
                        already selected */
                IF (((NUMBER := READY$TASK$TRAP(J)) = 0) AND
                        (NUMBER$IN$ROW$BACK(KBUS(J)) <= 0)) THEN DO;
                        /* select the task */
                        READY$TASK$TRAP(J) = DUMMY;
                        GENERATOR$NUMBER = J;
                        VSEM TRAP$SEM(J) RESET_;
                        RETURN STILL$GOING;
                        END;
                I = I + 1;
                VSEM TRAP$SEM(J) RESET_;
                END;
                RETURN STILL$GOING;
        END SEARCH$TRAP;

/* to find a backward substitution related task */
SEARCH$BACK: PROCEDURE BYTE;
DECLARE NUMBER INTEGER;
DECLARE I INTEGER;

        I = 0;
        DO FOREVER;
                IF ((NEXT$COLUMN$BACK < 1) OR (BACKWARD$DONE = TRUE))
                                THEN RETURN ENDED;
                IF I > SEARCH$LENGTH THEN RETURN TASK$NOT$FOUND;
                IF ( NEXT$COLUMN$BACK - I ) < 1 THEN RETURN TASK$NOT$FOUND;
                J = NEXT$COLUMN$BACK - I;
                PSEM B$SEM(POSN$IN$B$BEFORE$ORDERING(J)) SET_;
                IF (NUMBER := NUMBER$IN$ROW$BACK(J)) = 0 THEN DO;
                        NUMBER$IN$ROW$BACK(J) = DUMMY;
                        COLUMN$NUMBER = J;
                        VSEM B$SEM(POSN$IN$B$BEFORE$ORDERING(J)) RESET_;
                        RETURN STILL$GOING;
                        END;
                I = I + 1;
                VSEM B$SEM(POSN$IN$B$BEFORE$ORDERING(J)) RESET_;
                END;
                RETURN STILL$GOING;
        END SEARCH$BACK;
```

APPENDIX 6

PAPER TO APPEAR IN 'IEEE TRANSACTIONS ON COMPUTERS'

AN EFFICIENT PARALLEL ALGORITHM FOR THE SOLUTION OF LARGE SPARSE LINEAR MATRIX EQUATIONS

C.P. Arnold, Senior Member IEEE
M.I. Parr, Student Member IEEE
M.B. Dewe, Member IEEE

ABSTRACT

An algorithm for the parallel solution of large sparse sets of linear equations, given their factor matrices, is developed. It is aimed at efficient practical implementation on a processor of the Multiple Instruction Multiple Data stream (MIMD) type. The software required to implement the algorithm is described. In addition, the amount of memory necessary for data retention during execution is considered and related to that which is required on single processor systems. Hardware developed for the implementation of the algorithm is described. Bus contention for the system is outlined and shown to be insignificant. Possible bus contention problems for systems differing in the number of processors and speed of processing elements are also considered. A simulator modelling the execution of the algorithm on large systems has been implemented. The performance of the algorithm, in terms of execution speed enhancement relative to the theoretical maximum, is shown to be good.

INTRODUCTION

Considerable research has been directed at the parallel solution of linear equations in recent years. The authors' interest arises in power system transient stability analysis where the solution of a set of linear equations is required frequently, given a coefficient matrix which is modified only occasionally. The reforming of the factor matrices used in the direct solution is not therefore a major component of the overall computational effort required for the analysis. Hence,

work has been aimed at the parallel implementation of the substitution steps.

Transient stability analysis problems involve the solution of a large, sparse set of linear equations and, in addition, the solution of a set of differential equations which are, or are almost, independent. When a single processor is used the linear and differential equation solutions require similar portions of processing time. However, when parallel processors are employed the linear solutions become a bottleneck taking an increasing majority of processor time with increasing numbers of processors. Hence, there is a need for improved algorithms which increase the speed of solution of linear equations on parallel processors. Wing and Huang [1] have shown from a theoretical measurement that there is sufficient parallelism available within the solution of sparse linear equations to achieve considerable speed enhancement. If this level of performance can be realised in practice it will greatly improve upon presently available methods.

Single instruction multiple data stream (SIMD) processors, which take advantage of parallelism at the instruction level (e.g. vector operations), have been used to solve the linear equations describing power system networks. Happ et al [2] and Pottle [3] give examples of this approach. However, efficient application is difficult as the vectors contain a high proportion of zero elements. An alternative type of parallel processor, the multiple instruction multiple data stream (MIMD) processor can utilise parallelism at other than the instruction level. These are receiving the majority of current research interest and considerations made in this paper are restricted to this type.

To date, the approaches in developing a suitable algorithm fall into two distinct groups. One, which will be referred to as Block Schemes, is to divide the matrix of interest into blocks. References

[4]-[7] give examples using Block Schemes. The number of blocks is close to, or is simply related to, the number of processors. An alternative group of methods exploit the parallelism which is shown to exist when considering each individual element substitution during the solution, for example reference [1]. These methods shall be referred to as Elemental Schemes. With this approach the management costs can be great as the number of separate processes is much higher.

This paper describes the development of an algorithm based on an elemental scheme. A trade-off is made between utilisation of parallelism and management overhead to improve efficiency. The algorithm is suited to practical implementation and the requirements for realisation are given.

A simulation of the execution of the algorithm has been carried out on typical power system problems. The overheads due to bus contention are separately considered and are shown to be insignificant in the case of a multiprocessor developed at the University of Canterbury. The difficulties which could arise due to bus contention in other systems are described and potential solutions to the problem are briefly outlined. Results are presented and it is shown that efficiencies approach the theoretical maximum.

DEFINITIONS AND EXPLANATION OF TERMS

The LU factorisation method of solving a set of linear simultaneous equations of the form

$$Ax = b \tag{1}$$

is described in Appendix 1 and requires the construction of two factor matrices such that:

$$LU = A \tag{2}$$

This method was selected because

- it allows repeated solutions of 'x' for various values of 'b' without refactorising; and

- it provides an environment in which it is easier to visualise the step by step operation of the method relative to others, such as bifactorisation [8], which are equally applicable.

Because the L and U matrices are formed only occasionally, relative to the number of times they are used in substitution, no consideration is given to parallel triangulation.

The following terms are defined for consistency in the descriptions that follow.

Task graph:	a directed graph depicting the order of execution and times of synchronisation during the running of program code, i.e. data flow.
Nodes, edges:	the components of a task graph (as defined in ref. [1])
In,out-edges:	the edges entering and leaving a particular node, respectively
Dynamic:	varying at the time a program is executed
Process:	a block of program code executed serially by a single processor
Synchronisation:	this is necessary when a processor requires the results of one or more processes executed in other processors to continue its own execution
Task:	the program code associated with a node

At any time a task can have one of four states:-

not ready:	some necessary preceding results are not available so the task cannot be executed
ready:	all necessary results for initialisation are available and the task can begin execution
running:	the task is presently being executed
completed:	execution is completed and hence all results produced by the task are available

During either the L or U matrix substitutions each node can be uniquely identified by the location of the element in the matrix as required in a particular update operation. Hence, task graph models used in this paper utilise two indices to define nodes. In this manner the correspondence between the task graph and the L and U matrices is easily visualised.

Each node has the following properties:

In(i,j): the number of in-edges of node(i,j)

Pr(i,j): the set of all nodes preceding node(i,j)

Dynin(i,j): the number of elements of Pr(i,j) whose associated tasks do not have the completed state. Note that this is a dynamic property.

A general task graph for the solution of linear equations, using the L and U factors of A is given in figure 1. For consistency, the diagonal nodes in both the L and the U substitutions are included although one set will not normally require any actual program execution as they will be identity operations, i.e. divide by one.

BLOCK AND ELEMENTAL SCHEMES

The object of the algorithm is to exploit many processors efficiently in the solution of large sparse linear equations. The admittance matrices of specific interest contain around 90% to 99.5% zero elements and can have dimensions of many thousands.

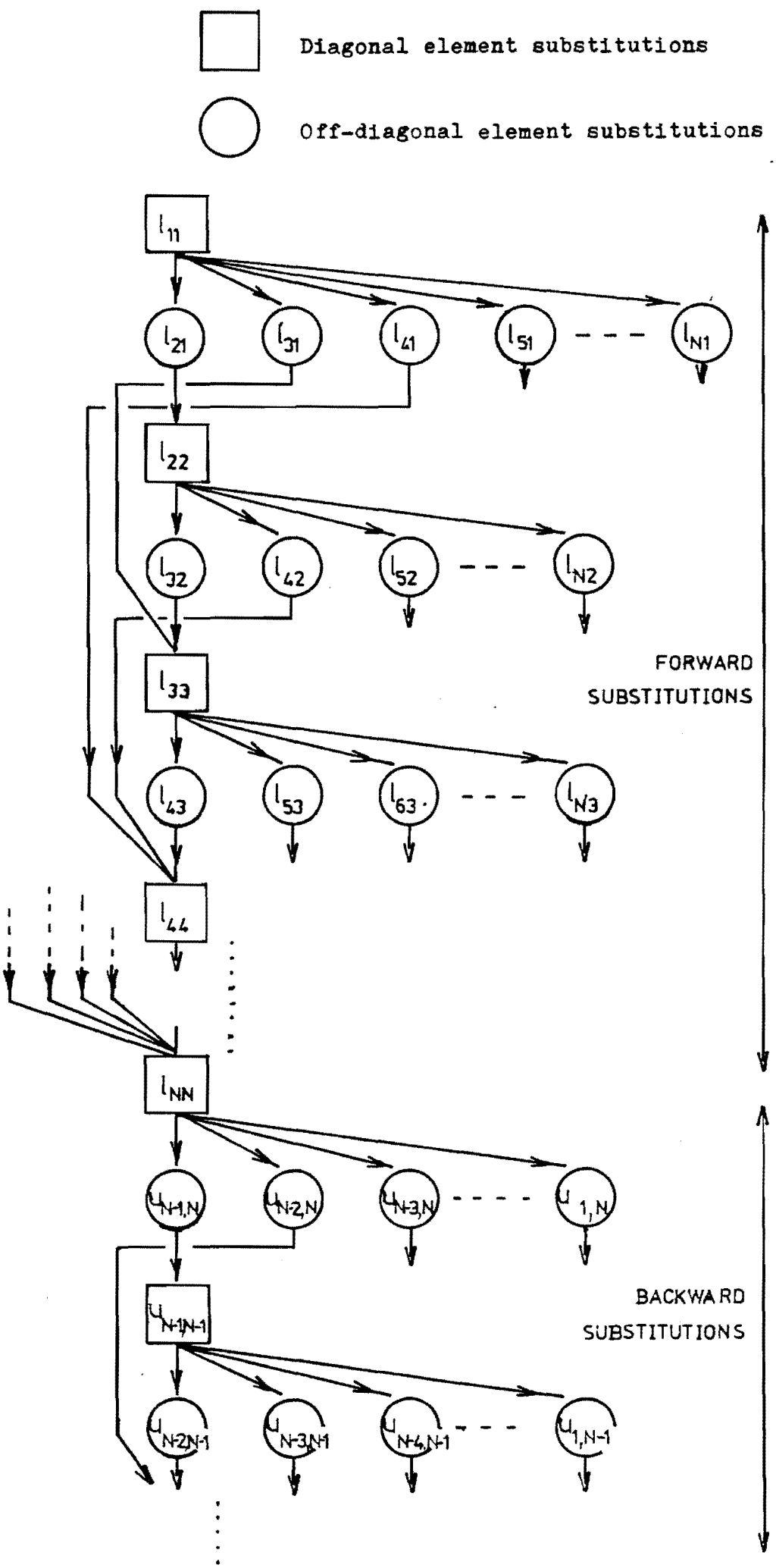
To create an algorithm, which will run efficiently in a parallel processing environment, the following requirements exist:

(1) that the problem is divided into a sufficient number of processes, which can be executed in parallel, and

(2) that the overheads involved in implementing the algorithm remain as low as possible.

As a problem is divided, producing more processes, the overheads associated with managing these processes increase. Hence, a trade-off

FIGURE 1 - Task Graph Depicting the LU Substitution Process



between (1) and (2) is likely.

In theoretical considerations it is useful to model the execution of a problem on a unit execution time task graph as in reference [1]. That is, transitions through all nodes are assumed to take the same time. Using a unit execution time task graph the optimal order of task execution can be determined before execution. In a practical implementation, unit execution time is not a valid assumption, as many factors affect the rate of execution of tasks. It is possible to force synchronisation of all tasks with the slowest task at each execution step, but this is undesirable because of the number of idle processors when task execution times vary significantly. A realistic approach is to use a dynamic method of task allocation, taking account of the actual execution rates of tasks.

Existing methods of solving linear equations on parallel processors fall into the two broad areas classified as block and elemental schemes. The distinction between these is now presented leading to the description of the new algorithm which is of the elemental type.

Block schemes are aimed at minimisation of the number of synchronisations required. Essentially this is accomplished by division of the L and U matrices into a small number of blocks. Figure 2 shows the possible distribution for a M-processor system.

By assigning fixed blocks to specific processors precedence properties between execution steps are easily defined. For example, consider the execution of the forward substitutions (L) in terms of an example task graph, figure 3. If processor 1 completes the substitutions in its associated triangular block, i.e. the left-hand column, then all other processors are in a position to make the substitutions in the first column of the block corresponding to the first two rows in figure 3. After this, processor 2 can make substitutions on its diagonal, and so on. Processes, in such a method, consist of all the substitutions in a block.

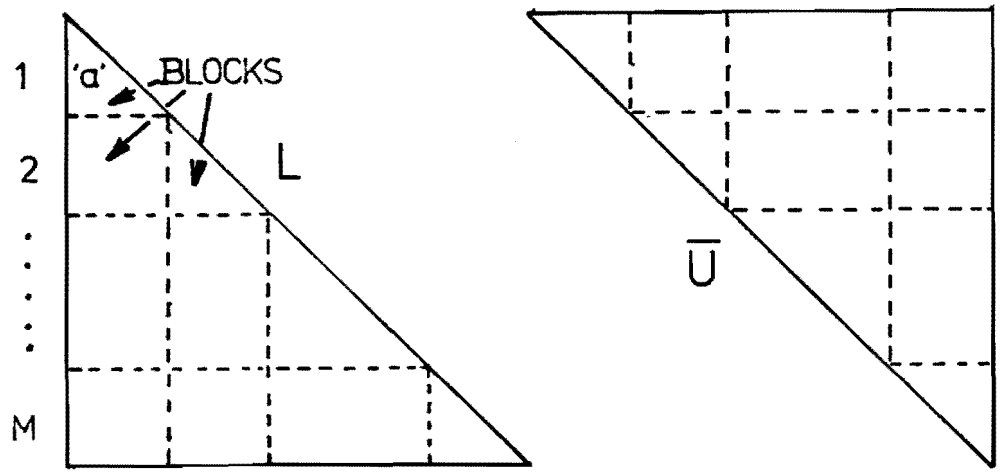


FIGURE 2 - Typical Distribution of Blocks Among Processing Elements for Block Schemes

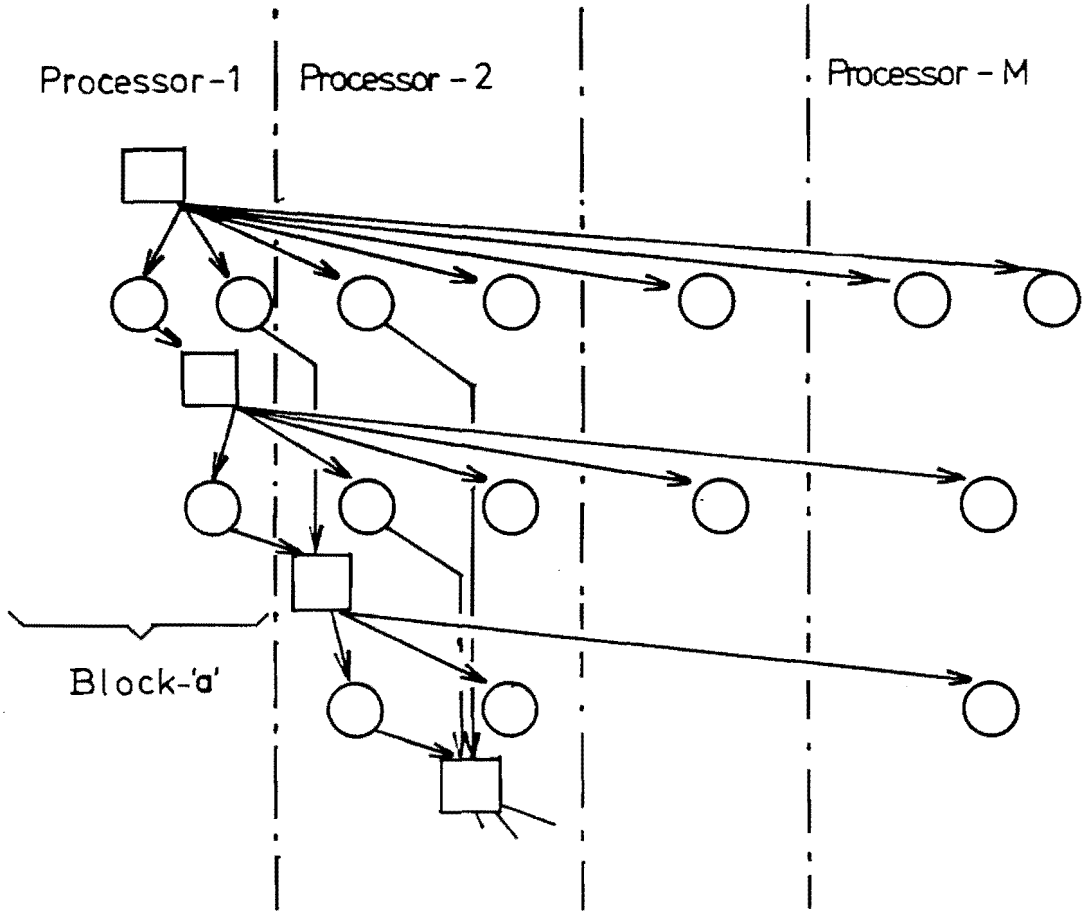


FIGURE 3 - Task Distribution among Processing Elements for Block Schemes

An example of an algorithm using an elemental scheme is given in reference [1]. The method is based on Hu's scheduling algorithm. In terms of a task graph this requires a search through all nodes (figure 1) to find 'ready' tasks which can be assigned to idle processors. In the event that there are many 'ready' tasks, distribution is determined by the 'depth' into the task graph and the number of out-edges. Although the description in reference [1] assumes unit execution time, the method can also be implemented dynamically. Ignoring the management required, this method indicates the maximum achievable efficiency of any parallel processing system when applied to the solution of linear equations, provided it is assumed that the nodes of the task graph (figure 1) are the smallest unit into which the problem can be divided.

THE NEW ALGORITHM

The rationale behind the algorithm presented here is the utilisation of the natural structure of the problem to identify processes which exploit sufficient parallelism without unreasonable management overhead.

- Problem Structure

A task graph depicting the substitution of the elements in the L and U matrices was given in figure 1. The following properties of LU substitutions are observed from the task graph.

$$Pr(i,j) = node(j,j);\ In(i,j) = 1 \qquad (3)$$

$$(i \neq j,\ \text{column } j,\ \text{all } i \text{ that exist})$$

$$Pr(i,i) = \{nodes(i,j)\} \qquad (4)$$

$$(i \neq j,\ \text{row } i,\ \text{all } j \text{ that exist})$$

The property given by (3) offers a considerable saving in the management required to determine 'ready' status of all off-diagonal tasks. That is, once a diagonal task has 'completed' status all the off-diagonal tasks in that column must be 'ready'. For this new algorithm, the basic process consists of a serial sequence of updates

using the diagonal element followed by the off-diagonal elements in a column of L or U.

Process management involves detecting which diagonal nodes have 'ready' status. This state is indicated, using the property given by (4), when all nodes in the row are 'completed' which corresponds to the time when, for a diagonal node (j,j):

$$Dynin(j,j) = 0 \quad (5)$$

To implement this management function, Dynin is stored as a vector with each element corresponding to a diagonal element in L or U. The initial value of Dynin is set by:

$$Dynin(j,j)initial = In(j,j) \quad \text{(for all j)} \quad (6)$$

During execution the contents of Dynin are updated after each node is substituted. For node (i,j)

$$Dynin(i,i) = Dynin(i,i) - 1 \quad (7)$$

In a practical implementation, the functions of updating and observing the state of the Dynin vector ((5) and (7)) can be handled in a variety of ways. The method investigated involves each processor supervising itself by searching Dynin for ready tasks. One alternative is to assign a specific processor to searching Dynin.

- Process Definition

For the implementation investigated all processors are assumed to be set up with identical local program code. A process consists of the three sections shown in figure 4 which are described below using the symbols introduced in Appendix 1.

. Search

The objective here is to find a 'ready' diagonal node. This is achieved by observation of the Dynin vector awaiting condition (5). Having located an element satisfying this condition the column associated with that element is selected for substitution. To inform other processors that the column has been selected a known negative value is substituted

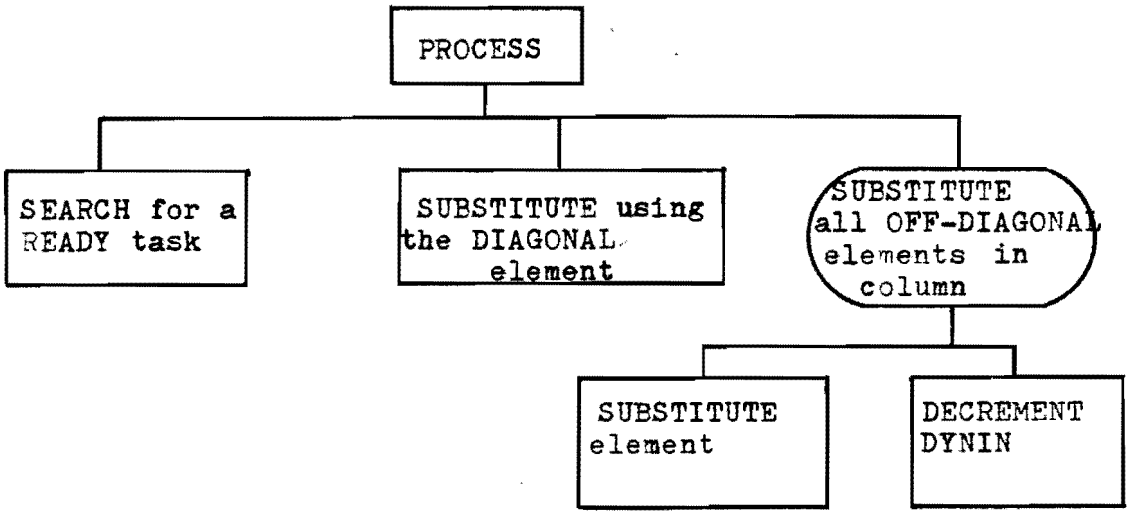


FIGURE 4 - Basic Process for the New Algorithm

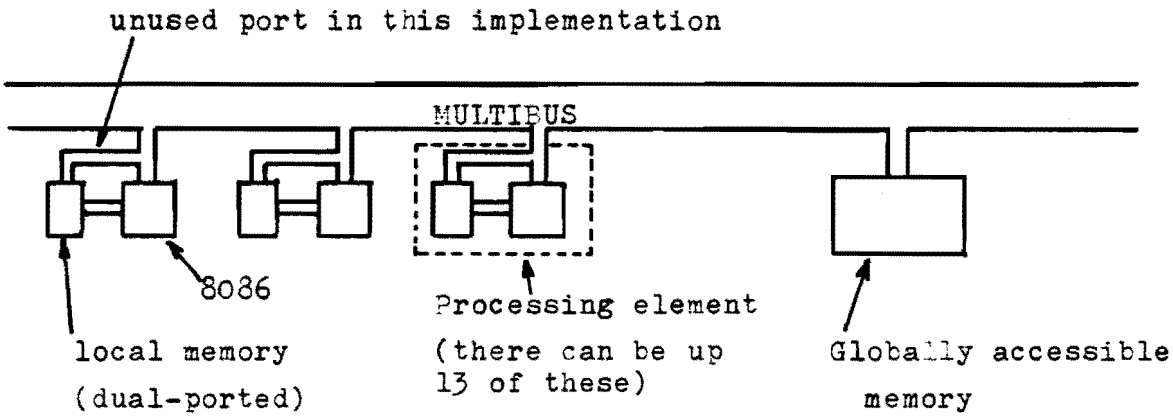


FIGURE 6 - Practically Implemented Multi-Microprocessor System

in Dynin indicating that the state has changed to 'running'.

. Diagonal Update

For any element 'i' in Bg' the diagonal update involves:

for L : b'(i) = Bg'(i)/l(i,i)

for U : x(i) = Bg'(i)/l

. Column Substitutions

for L : for all non-zero elements in column j

Bg'(i) = Bg'(i) - b'(j)*l(i,j)

Dynin(i,i) = Dynin(i,i) - 1

for U : for all non-zero elements in column j

Bg'(i) = Bg'(i) - x(j)*u(i,j)

Dynin(i,i) = Dynin(i,i) - 1

The contents of Dynin must be reinitialised before both the forward and backward substitution steps.

- Use of Semaphores

Each element of Bg' and Dynin is a resource common to all processors and can be updated by any processor. To maintain security during updates a semaphore must be associated with each element of Bg' and Dynin. For practical implementation, the use of semaphores requires a bus locking capability. That is, a single processor must be able to stop any other accesses to the location containing the semaphore while it checks and sets the value.

BUS CONTENTION AND PRACTICAL IMPLEMENTATION

In the implementation of an algorithm on a parallel processor limitations to performance arise in two related areas.

. inherent lack of parallelism in the algorithm.

Details of this overhead are discussed in the simulation description which follows, and

. hardware contention problems. These arise due to common utilisation of hardware resources. In this context the problem is most apparent where

processors share common memory.

Three types of information which must be stored are categorised as follows:-

. Definitely Global - data which must be available to all processors. The contents of the Bg' vector, the Dynin vectors, and their associated semaphores are included.

. Preferably Global - data which would most conveniently be available to all processors. The contents of the vectors in figure 5 except those which are definitely global fall into this category. If this information is not globally available then either it must be stored a number of times elsewhere or individual processors must be assigned specific tasks before execution commences.

. Local - information such as program code, intermediate results of operations, and stack contents, which need only be available to individual processors.

For a specific hardware implementation the choice between storing preferably global data in locally accessible or globally accessible memories will be the result of a consideration as to whether or not the bus contention problems involved when using globally accessible memory outweigh the problems of local storage.

A multiprocessor using a single interconnecting bus (INTEL MULTIBUS) has been developed at the University of Canterbury as part of this project. Up to 13 INTEL 8086 microprocessors at present can be connected in the structure illustrated in figure 6. Each processor has access to its own local memory and to global memory. For this system, as is outlined below, the bus contention problem results in an insignificant reduction in processing speed even when preferably global data is stored in globally accessible memory.

For a typical update step in the substitution process, consisting of a complex multiplication and addition, four 32-bit real values are read

from global memory and two are written. Also at least three accesses are made to set and reset semaphores and for each task two index values are read. Using 16-bit words and allowing for a few extra semaphore checking accesses, about 20 transfers via the single bus will be required, i.e. 12uS with a 600 nS typical access time. As shown in Table 1 the total execution time will be approximately 1400 uS. Therefore, a figure of about 1% total bus utilisation per processor may be expected.

Using 10 processors with 1% bus utilisation each, the possibility of a bus conflict at each access is close to 10%. Therefore, an increase of about 10% in average access time to global memory is expected and, as this represents approximately 1% of all accesses, a loss in performance by each processor, and of the system as a whole, of about 0.1% is likely. Extending this towards 100 processors bus contention overhead would increase to about 10% as, not only is the bus more likely to be busy, but also the number of accesses pending from other processors, on average, is greater. For many more than 100 processors bus saturation, i.e. 100% bus utilisation, would occur rendering any further increases in the number of processors useless.

To consider that this low level of bus contention would be typical of all processor types is ill-advised. In the execution of floating-point operations the microprocessor is relatively slow. This factor, as well as the restricted number of processors of the University of Canterbury system, results in a low overhead. Using faster processing elements a single bus could become very restricting and performance improvement would require the implementation of more complex bus structures. This problem has been addressed in many recent publications, an example of which is reference [9]. It is not the intention here to consider optimal bus structures except to suggest that methods are available for the distribution of global memory on multiple buses which individual processors could switch between.

The extra storage required for information specifically related to multiprocessing is a cost to be taken into account in the design of parallel processing systems. Figure 5 illustrates the vectors used and they are defined in Appendix 2. Comparison is made with those vectors needed on a single processor system to give some idea of the additional memory required. For example, a 1000 bus system with a sparsity coefficient of 0.95 would require 373,000 bytes for data storage using this algorithm as opposed to 264,000 for a serial implementation. Power system problems of this size may well have sparsity coefficients in the order of 0.99 where the figures are 93,000 and 64,000 bytes respectively. Each processor requires in practice about 15,000 bytes of local memory. Memory to cope with this order of problem size is readily available.

SIMULATION

Because the new algorithm requires considerable management overhead, and does not exploit parallelism to the same extent as Hu's method its performance is expected to be worse than that achieved in [1] where Hu's method was modelled without management overhead.

The objective in using a simulation of the execution of the algorithm is to evaluate its expected performance on a machine with a generalised architecture. The simulator takes account of the major overheads involved and produces results expressing performance as a function of the number of processors.

As the algorithm is dynamic it would be unrealistic to model the execution using fixed basic operation processing times. The order of process initialisations is determined by randomly occurring events which cannot be modelled accurately in the simplistic manner used in references [1] and [2]. Hence, a detailed model is used which emulates execution as a function of time through a complete substitution step.

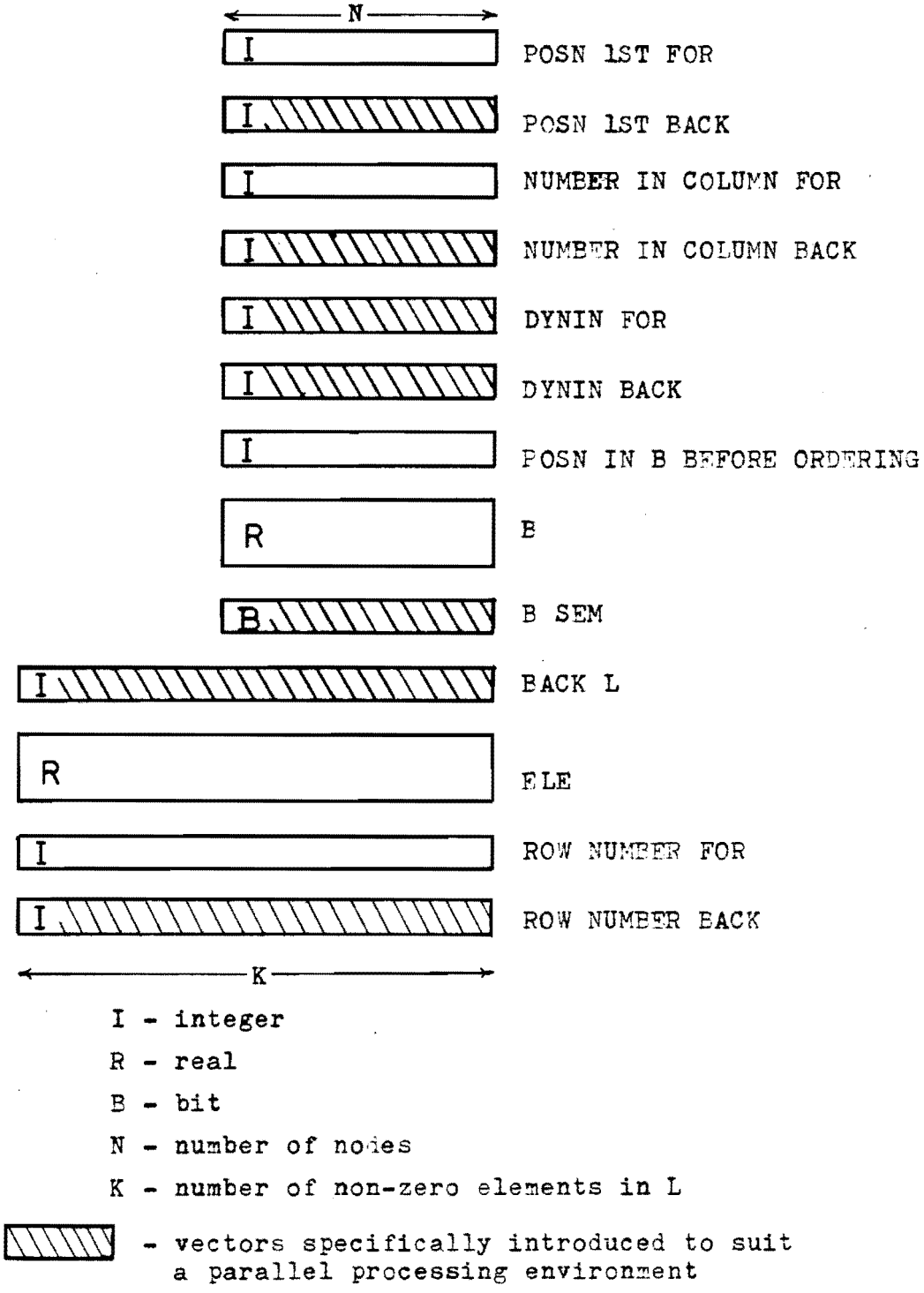


FIGURE 5 - Vectors and Storage Requirements for the New Algorithm

- Overheads

An overhead is any event resulting in a difference between the speed of a parallel processor and the product of the speed of a single processor and the number of processors.

For an LU substitution sequence overheads can arise in the following areas:

1. An increase in the number of elements in L and U, due to special reordering, which results in extra substitutions and hence additional processing. Special reordering refers to, for example, the formation of suitable blocks in the factor matrices used in Block schemes
2. Additional computation due to functions related to multiprocessing, e.g. searching for 'ready' tasks.
3. Processors idling because the result of another task or tasks is required before any further processes can begin useful execution.
4. Processors idling because the common bus (or buses) shared by the processors becomes busy resulting in processors having to wait for access.

- Features of the simulator used

Processor operation times are selected, once for each operation during execution, from a rectangular distribution. Each basic operation has a separate distribution allowing a variety of processor types to be modelled.

Overheads modelled include:

1. Extra processing in a multiprocessing environment
   .Search for vacant columns
   .Setting of semaphores during the search
   .Setting of semaphores during update operations
   .Decrementing the elements of Dynin
2. Idle processors
   .No tasks available
   .An element is presently being updated by another processor

It is assumed that special reordering in the formation of the L and U matrices is not employed. Bus conflicts are not modelled as they would vary depending on the bus structure. As outlined in the previous section for the multiprocessor developed at the University of Canterbury the overheads due to bus conflict are insignificant.

To enable comparison of the relative effects of the various overheads, provision is made for individual measurement. The overheads are categorised as:

1. No tasks available
2. Search for vacant columns, including semaphore updates
3. Decrementing Dynin including semaphore updates
4. Simultaneous updates of elements of the Bg' vector

The number of columns which each processor can look ahead for a 'ready' task is limited. In practice this will be an efficient approach as the most likely tasks to be ready will be nearest to those just 'completed'. On reaching the limit the processor restarts the search at its original starting point which may, by this stage, contain a 'ready' task.

- Simulation Results

Results are presented for four sets of conditions, i.e. two networks and two processor types.

The networks used are:

- 400 bus network with randomly distributed elements, 99.2% sparse (similar to that used in reference [1]).

- 2000 bus network with randomly distributed elements, 99.5% sparse

The processors modelled are:

- a 16-bit microprocessor (INTEL 8086)

- a 48-bit mainframe (BURROUGHS B6700)

These were selected as their speeds are expected to give results indicative of two classes of processor. Table 1 indicates the speed

data used, based on references [10] and [11]. No account has been taken of the available bus structures for the two processor types.

TABLE 1 - Average Operation Execution Times for Two Processor Classes

OPERATION	8086(uS)	B6700(uS)
substitution of diagonal elements	1172	44
substitution of off-diagonal elements	1400	76
search each element of Dynin	20	16
semaphore set and reset	18	10
decrement Dynin element	10	6

Figure 7(a) illustrates the performance of both processor types as a function of the number of processors. Note that the B6700 performance is consistently about 20% below that of the microprocessor. Changing the variance of the speed data was found to have no significant observable effect on the position of the performance curves. It was also found that performance is insensitive to the exact form in both sparseness and the distribution of elements. Included in figure 7(a) is a curve corresponding to the results obtained using Hu's scheduling algorithm [1] for a similar network. This allows some comparison with the maximum achievable efficiency.

For low numbers of processors the management overheads dominate, while for higher numbers the lack of available processes becomes more significant. This is illustrated in figures 7(b), (c) where the categories of overhead defined previously are employed. The relative effects of the various types of overhead are indicated. The overhead

Note: Figures 7 and 8 were presented in Chapter 8 with the following numbering:

7(a) - 8.3

7(b) - 8.5

7(c) - 8.6

8(a) - 8.4

8(b) - 8.7

8(c) - 8.8

due to management is greater for the faster processor. The reason is as follows. Instructions to be executed include firstly, useful processing and, secondly, management processing required only in a multiprocessing environment. The former predominantly involves the execution of floating point instructions, while the latter does not require the use of such instructions. Relative to the microprocessor, the mainframe processor is much faster at executing floating-point instructions but similar in speed for other instructions. Hence, the mainframe spends a greater proportion of its time in management type execution.

Similar results were obtained for the larger network and these are shown in figure 8(a)-(c). Note that for the microprocessors the speed rounds off at around 100 times that of a single processor. It must be remembered that bus structures which efficiently support such systems have not been investigated.

CONCLUSIONS

An algorithm for the solution of large sparse linear equations on a multiprocessor has been presented. The algorithm is designed so that it achieves a suitable compromise between exploitation of parallelism and extensive overhead in implementation.

Results of a simulation of the execution of the algorithm indicate that the losses in performance, due to a reduction in exploitation of parallelism relative to other methods, are low.

The overheads introduced to implement the method have different effects on performance for different classes of processor. The mainframe processor simulated had a performance around 20% below that of the microprocessor, where performance is measured relative to a single processor of the same type. In all cases investigated the dominant overhead for large numbers of processors is a lack of available processes which can be executed in parallel.

For microprocessor based systems with relatively few processing

elements bus contention is not a restrictive problem even using a simple bus structure. However, more complex bus structures may be necessary to efficiently implement the algorithm on systems with more and/or faster processing elements. There is also scope for redistribution of data to local storage elements reducing the number of accesses to global memory.

This algorithm offers a practical solution to the major problem existing to data in the parallel implementation of transient stability analysis programs. That is, the bottleneck which arises in the solution of the linear equations, when combined with the solutions of independent differential equations, is removed.

APPENDIX 1

Solution of Sparse Linear Equations

The LU method of solution of linear equations [12] involves expressing the coefficient matrix, A, as the product of two factor matrices, L and U. i.e. for an equation of the form

Ax=b where x and b are vectors whose elements will be referred to as x(i) and b(i) with index i, form L and U such that

A=LU

L is a lower triangular matrix, i.e. it has no non-zero elements above the diagonal.

U is an upper triangular matrix with unity elements on its diagonal.

Once the L and U factors are found by triangulation, the solution steps in finding x are

1. to find an intermediate vector b' by back-substitution

   Lb'=b ; and similarly

2. to find the solution x

   Ux=b'

A general element of 'b' as it is transformed to 'x' will be referred to as Bg' and elements of L and U by the lower case equivalent, e.g. l(i.j).

APPENDIX 2

Definition of vectors used in algorithm implementation:

ELE : The elements of the L and U matrices

POSN 1ST FOR : position within ELE of the diagonal element in each column for forward substitutions

POSN 1ST BACK : similar for back substitutions, but off-diagonal

NUMBER IN COLUMN FOR : number of elements in each column for forward substitutions

NUMBER IN COLUMN BACK : similar for back substitutions

DYNIN FOR : number of elements in each row for forward substitutions

DYNIN BACK : similar for back substitutions

B : elements of Bg'

POSN IN B BEFORE ORDERING : maps the present values of the B vector to the positions they were in before ordering, where the ordering was done for any reason.

B SEM : semaphores corresponding to both elements of B and columns. Separate sets of semaphores could be used.

BACK L : vector to map to indices of ELE for back substitutions

ROW NUMBER BACK : row numbers of elements within ELE

Affiliation of Authors

Dr. Arnold and Mr. Parr are, and Mr. Dewe was, with the Department of Electrical Engineering, University of Canterbury, Christchurch, New Zealand. Mr. Dewe is now with Wormald Vigilant Limited, Christchurch, New Zealand.

Acknowledgement

The authors wish to acknowledge the support of the New Zealand University Grants Committee.

REFERENCES

[1] Wing, O., and Huang, J.W. "A Computational Model of Parallel Solution of Linear Equations", IEEE Trans. on Computers, vol. C-29, pp. 632-638, July 1980.

[2] Happ, H.H., Pottle, C., and Wirgau, K.A. "Parallel Processing for Large Scale Transient Stability", IEEE Canadian Conf. Comm. and Power, 78 CH 1373-0 REG 7, pp. 204-207, 1978.

[3] Pottle, C. "The use of an Attached Scientific ('array') Processor to Speed up Large-Scale Power Flow Simulations", IEEE Int. Conf. on Circuits & Computers, ICCC 80, Port Chester, NY, Oct. 1980.

[4] Wallach, Y. and Conrad, V. "On Block-parallel Methods for Solving Linear Equations", IEEE Trans. on Computers, vol. C-29, pp. 354-359, May 1980.

[5] Brasch, F.M., Van Ness, J.E., and Sang-Chul Kang. "Evaluation of Multiprocessor Algorithms for Transient Stability Problems", EPRI EL-947 Technical Planning Study 77-718, Nov. 1978.

[6] Fong, J. and Pottle, C. "Parallel Processing of Power System Analysis Problems via Simple Parallel Microcomputer Structures". IEEE Trans. on Power Apparatus & Systems, vol. PAS-97, pp. 1834-1841, Sept/Oct. 1978.

[7] Hatcher, W.L., Brasch, F.M. and Van Ness, J.E. "A Feasibility Study for the Solution of Transient Stability Problems by Multiprocessor Structures", IEEE Trans. on Power Apparatus & Systems, vol. PAS-96, pp. 1789-1796, Nov/Dec. 1977.

[8] Zollenkopf, K. "Bifactorisation: Basic Computational Algorithm and Programming Technique". In 'Large Sparse Sets of Linear Equations'. pp. 75-96, Academic Press, 1971.

[9] Patel, J.H. "Performance of Processor-Memory Interconnections for Multiprocessors", IEEE Trans. on Computers, Vol. C-30, pp. 771-780, Oct. 1981.

[10] 8086 Microprocessor Seminar Notes, INTEL Corp., 1980.

[11] B6700 Timings for Selected Language Constructs (Relative to Mark 2.6 Software Release); Burroughs Corp. Form No. 5000854, February, 1975.

[12] Brameller, A., Allan, R.N., and Haman, Y.M. "Sparsity", Pitman Publishing, 1976.